

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Vignelli — Automated Design Guidance for Developers

Author:
Simon STUCKEMANN

Supervisor:
Dr. Robert CHATLEY
Second Marker:
Prof. Duncan GILLIES

Submitted in part fulfilment of the requirements for the degree of
Master of Engineering in Computing

June 16, 2015

*I like design to be
semantically correct,
syntactically consistent, and
pragmatically understandable.*

*I like it to be visually powerful,
intellectually elegant, and
above all timeless.*

MASSIMO VIGNELLI

Abstract

We present *Vignelli*, an IDE plugin that helps developers improve their software designs by accelerating the software design feedback loop. By continuously observing the code that the developer is writing in their IDE, the plugin is able to detect design flaws such as “train wrecks”, “direct uses of singletons” and “long methods”, inform the developer about these flaws, and assist in the refactoring of the code to move towards a better design for some of these problems.

Existing tools and techniques addressing this issue typically feature very slow feedback loops which result in only very slow learning progress. Our tool analyses the structure of the code being edited and its relationship to other existing modules in the project. Our results show near perfect accuracy for the detection of “train wrecks”, and 92.68% precision and 82.61% sensitivity for the detection of “direct uses of singletons”, whilst giving real-time feedback while the developer is working. Furthermore, *Vignelli* successfully assists in the refactoring of 50% of “train wrecks” and 73.68% of “direct uses of singletons”.

Acknowledgements

I would first like to thank my supervisor, Dr Robert Chatley, for his constant support throughout this project. His ideas, advice and recommendations have helped bring *Vignelli* to life.

I would also like to thank Dimitry Jemerov of JetBrains, whose incredibly helpful and quick answers have enabled me to write a better IntelliJ plugin than would otherwise have been possible. Further, I would like to thank Ella Su, who has found and reported countless bugs in *Vignelli* throughout this project.

I would also like to thank my family and friends for their tremendous support not only during this project but throughout my four years at Imperial. Finally, I wish to take this opportunity to thank Rose for her constant support and encouragement. I would never have made it without you.

Contents

1	Introduction	5
1.1	Motivation	5
1.2	Objectives	6
1.3	Contributions	6
2	Background	7
2.1	Feedback Loops	7
2.2	Importance of Good Software Design	8
2.3	Design Patterns	9
2.3.1	Singleton Pattern as an Example of a Creational Pattern	10
2.4	Code Smells	12
2.4.1	Method Call Chains and Train Wrecks	12
2.4.2	Direct Use of Singleton	17
2.4.3	Long Method	19
2.5	Learning About Software Design	21
2.6	Development Techniques	22
2.6.1	Refactoring	22
2.6.2	Manual Code Review	23
2.6.3	Testing	25
2.7	Tools	25
2.7.1	Integrated Development Environments (IDEs)	25
2.7.2	FindBugs	27
2.7.3	Checkstyle	28
2.7.4	PMD	29
2.7.5	JDeodorant	30
2.8	Code Structure Analysis	32
2.8.1	Structure and Collaboration Templates to Identify Design Patterns	33
2.8.2	Finding Bug Patterns Using FindBugs	34
2.8.3	Matrix-Guided Directed Search for Design Patterns	34
2.9	Metrics-Based Analysis	35

2.9.1	Commonly Used Metrics	35
2.9.2	Empirical Detection of Long Method Smell	36
2.9.3	Combinatorial Software Design Optimisation	38
3	Design and Implementation	39
3.1	User Interface Design	39
3.1.1	Observation Mode	39
3.1.2	Refactoring Mode	40
3.1.3	Plugin Interaction and Developer Experience Level	41
3.2	System Architecture	41
3.2.1	IntelliJ Program Structure Interface (PSI)	43
3.2.2	Identification of Problems	45
3.2.3	Launching Refactorings	46
3.2.4	Refactorings and Refactoring Steps	47
3.2.5	Refactoring Step Goal Checkers	48
3.2.6	Handling Refactoring Step Results	50
3.2.7	Undo Operation	51
3.2.8	User Interface Coordination	52
3.3	IntelliJ IDEA Plugin Development	54
3.3.1	IntelliJ Refactoring Options	54
3.3.2	Goal Checkers and PSI Tree Changes During Multi-Step IntelliJ Refactorings	55
3.4	Development Process	56
3.4.1	Continuous Integration	56
3.4.2	Testing	57
3.4.3	Documentation	60
4	Identification and Eradication of Train Wrecks	61
4.1	Identification	61
4.1.1	Simple Approximation: Multiple Call Chains	61
4.1.2	Train Wreck Classification in the Context of Fluent Interfaces	62
4.1.3	Computing the Object Navigation Structure	63
4.1.4	Static Approximation for Object Navigation Structure	65
4.1.5	The void Type	66
4.1.6	Support for Static Calls	68
4.1.7	Train Wrecks in Practice: External Libraries	70
4.2	Refactoring	70
4.2.1	Example Step-By-Step Walkthrough	70
4.2.2	Refactoring Entry Points	73

4.2.3	Inline Variable	74
4.2.4	Extract Method Refactoring Step	75
4.2.5	Introduce Parameters for Members	77
4.2.6	Move Method Refactoring Step	79
4.2.7	Rename Method Refactoring Step	81
4.3	Refactoring Train Wrecks Involving Fluent Interfaces	82
4.3.1	Critical Call Chains	83
4.3.2	Additional Step: Introduce Parameter for Critical Call Chain	84
5	Identification and Eradication of Direct Singleton Retrievals	87
5.1	Identification	87
5.2	Refactoring	88
5.2.1	Example Step-By-Step Walkthrough	89
5.2.2	Refactoring Entry Points	91
5.2.3	Convert to Constructor-Initialised Field	91
5.2.4	Introduce Parameter in Constructors	94
5.2.5	Type Migration to Use Interface Wherever Possible	95
5.2.6	Finding Existing Interfaces	95
5.2.7	Perform Type Migration to Sufficient Interface	97
5.2.8	Extract Interface and Use Wherever Possible	99
6	Long Method Detection	101
6.1	Implementing Existing Metrics-Based “Long Method” Detection	101
6.2	Exploring the Validity of the Probabilistic Model	102
6.3	Towards a Generalised Probabilistic Model	103
6.3.1	Method Metrics Collection Using <i>Vignelli</i>	104
6.3.2	“Long Method” Classification Collection	104
6.3.3	Analysis Scripts	104
6.3.4	Preliminary Results and Outlook	105
7	Evaluation	107
7.1	Train Wrecks	107
7.1.1	Identification of Train Wrecks	107
7.1.2	Refactoring of Identified Train Wrecks	110
7.2	Direct Use of Singleton	113
7.2.1	Identification of Direct Uses of Singletons	113
7.2.2	Refactoring of Identified Direct Singleton Uses	115
7.2.2.1	Goal Checker Check All Limitation	117
7.3	User Testing	117

7.3.1	Iterative Improvements Through Continuous Feedback	117
7.3.2	User Study	118
7.4	Performance Testing	120
7.4.1	Performance Test Results	121
7.5	Stability	122
7.6	Learning Improvement	123
8	Conclusions	124
8.1	Core Achievements	124
8.2	Future Work	125
A	Test Setup	127
A.1	Hardware Configuration	127
A.2	Software and Environment Configuration	127
	Bibliography	128

Chapter 1

Introduction

1.1 Motivation

As software becomes more and more pervasive, it also becomes increasingly complex. Modern software systems not only have to integrate with other systems but are also required to be extensible and easy to change to allow future changes in requirements. If not managed well, this complexity can easily manifest itself in a design that can be described as rigid, fragile and immobile. On the other hand, software that is designed well exhibits the opposite characteristics: these systems are usually extensible, flexible and feature components that can easily be reused elsewhere.

Good software design practices are therefore essential skills for software engineers and improving these skills is a continuous process. The most effective way software engineers learn about design is through personal experience. Another very effective, but labour-intensive way to improve one's design sensibilities is to be mentored by a more experienced engineer who is able to give design guidance and who can explain his or her viewpoint using their previous experiences. Other ways to learn about software design include the use of tools and techniques such as books and articles, university courses, and group discussions.

Aside from mentoring, all of the approaches mentioned above generally only begin to show their benefits after longer periods of trial and error. Putting the newly-learnt design concepts into perspective and fully understanding them to a level that is sufficient for making educated design decisions, requires experience with the design choices in real-world applications. This means that the feedback loop is generally slow and learning takes a long time. Being mentored by a more experienced software engineer is very labour-intensive but does not share the drawbacks of slow feedback as experiences can be shared and related to real-world code directly and easily.

Fast feedback loops such as this one involve giving feedback on smaller chunks of work, but more frequently. This has several benefits: firstly, feedback on smaller chunks of work is easier to understand. Secondly, by getting feedback more often, it is easier to learn from mistakes quickly and make adjustments. Thirdly, by repeatedly getting the same feedback on small chunks of work, we are able to practise correcting our mistakes more often, thus improving our skill set.

Fast feedback loops are already common-place in some areas of computer science. For example, modern integrated development environments (IDEs) feature continuous compilation of code. While the developer is typing code, the IDE continuously compiles it and underlines any static errors. The developer can then fix these errors almost immediately.

However, as we have seen above, software design feedback loops are generally slow or labour-intensive. In this project, we aim to accelerate this feedback loop.

1.2 Objectives

The goal of this project is to develop a tool that helps developers to improve the design of their software by:

- Accelerating the software design feedback loop by continuously giving developers feedback on their design decisions in real time
- Assisting developers in refactoring their code to eradicate detected problems
- Providing extension points to support identifying other design problems in the future

1.3 Contributions

In this dissertation we present *Vignelli*, an IntelliJ IDEA plugin that detects design flaws in code, informs developers about these design flaws, and assists in the refactoring process towards a better design. The main contributions are summarised below:

- *Vignelli* accelerates the software design feedback loop by continuously observing the code the developer is currently working on.
- The plugin relates software design theory to how the concepts are applied in practice. This is done by explaining design flaws in the context of the application being developed by including class names and code in the detailed descriptions of problems and refactorings.
- Once a design flaw has been identified, the tool guides developers through a number of refactoring steps towards a better design. Each refactoring step is explained in detail and can be performed using IntelliJ’s refactoring capabilities or even manually.
- *Vignelli* is able to identify and assist in the refactoring of so-called “train wrecks” which are method call chains that ask for data from objects that are unrelated to the current class (see section 2.4.1). Early experiments have shown that we are able to detect train wrecks with near-perfect accuracy (see section 7.1).
- The plugin is able to identify and assist in the refactoring of so-called “direct uses of singletons” which occur when a class introduces coupling by retrieving a singleton instance directly via the singleton’s instance retrieval method (see section 2.4.2).
- We have built a system that is able to be extended easily as illustrated by the addition of a metrics-based search for so-called “long methods” (see section 2.4.3).

Chapter 2

Background

2.1 Feedback Loops

Feedback loops are an integral part of producing quality software. For example, writing a piece of code and subsequently running the compiler to find out whether it can be compiled or not constitutes a feedback loop. Successful compilation of the code means positive feedback to the programmer that the last piece of work does not contain any static errors. A failed compilation, on the other hand, gives negative feedback. No matter which outcome, this feedback is useful as it can inform the developer's next actions, e.g. fixing any existing syntax errors.

In his article *Frequency Reduces Difficulty*, Fowler notes that there appears to be an exponential relationship between the time between actions (e.g. running the compiler) and the amount of pain that is involved in fixing any issues that may occur (see figure 2.1) [1].

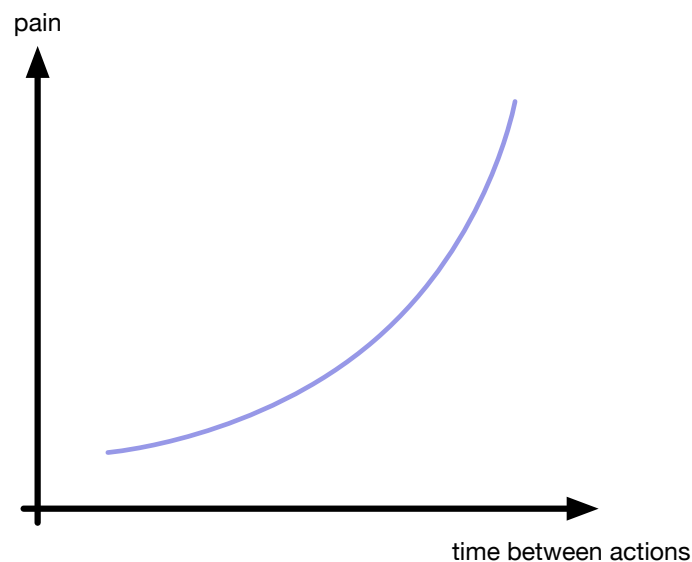


Figure 2.1: Waiting longer between actions leads to exponentially more pain

The longer we wait before trying to compile, the more likely it is for the compilation to fail. More changes also make finding bugs and mistakes more difficult.

The solution to this problem is to increase the speed of the feedback loop. According to Fowler, this has three main effects:

1. Splitting the work into smaller chunks makes it easier to understand and manage the changes
2. The reduced time for feedback means that learning and adjusting plans if necessary can happen much faster than before
3. Working on small, but difficult pieces of code more frequently means that we practice the task more, thus improving our skill set

Most integrated development environments (IDEs) (see section 2.7.1) continuously compile the source code that is being edited. Any static errors are then highlighted instantly (e.g. with red underlines) which means that developers generally fix compilation problems almost immediately.

Fast feedback loops are at the heart of many of today's most popular development methodologies. For example, test-driven development (TDD) emphasises a fast feedback loop to minimise the chances of introducing bugs. Once one test has been written the goal is to make it pass, without writing further tests or functionality. After refactoring (see section 2.6.1), this loop starts again. Agile development processes revolve around constant customer feedback that allows software teams to adjust quickly by learning from mistakes. Continuous delivery techniques are based on the concept that “automation is the key to fast feedback” [2, p. 14] and reduce the pain of deployment by submitting small incremental changes.

When it comes to software design (see section 2.2), fast feedback loops are less common and more difficult to achieve. Although it is desirable for software engineers to get fast feedback on their design choices in order to avoid accidental complexity, this is often a manual process involving code reviews (see section 2.6.2). *Vignelli* tries to make this process easier by automatically identifying potential design problems as the developer is writing code, thus immediately giving feedback on the design choices. The tool then suggests ways to refactor the code and explains the motivations for doing so, reducing the amount of pain involved in making these changes at an early stage before further complexity is added.

2.2 Importance of Good Software Design

According to McConnell, one of the main *technical* reasons why software projects fail is uncontrolled complexity. This happens when “software is allowed to grow so complex that no one really knows what it does” [3]. Ultimately, the management of this complexity is essential to designing software.

To be able to manage complexity adequately, it is important to understand the symptoms of a bad design. R. C. Martin summarises these as follows: [4]

Rigidity “Rigidity is the tendency for software to be difficult to change, even in simple ways. Every change causes a cascade of subsequent changes in dependent modules. What begins as a simple two day change to one module grows into a multi- week marathon of change in module after module as the engineers chase the thread of the change through the application.”

Fragility “Closely related to rigidity is fragility. Fragility is the tendency of the software to break in many places every time it is changed. Often the breakage occurs in areas that have no conceptual relationship with the area that was changed. [...] As the fragility becomes worse, the probability of breakage increases with time, asymptotically approaching 1. Such software is impossible to maintain.”

Immobility “Immobility is the inability to reuse software from other projects or from parts of the same project. It often happens that one engineer will discover that he [or she] needs a module that is similar to one that another engineer wrote. However, it also often happens that the module in question has too much baggage that it depends upon. After much work, the engineers discover that the work and risk required to separate the desirable parts of the software from the undesirable parts are too great to tolerate.”

On the other hand, good software designs are extensible, flexible and feature reusable components.

Unfortunately, as Martin points out in his book *Clean Code: A Handbook of Agile Software Craftsmanship*, “if you have been a programmer for longer than two or three years, you have probably been slowed down by messy code. [...] Every addition or modification to the system requires that the tangles, twists, and knots be ‘understood’ so that more tangles, twists, and knots can be added” [5].

As code evolves and more features are added, it is easy for accidental complexity to enter the system. Without good design, code often ends up in a “Big Ball of Mud”, a “casually, even haphazardly, structured system” [6] that is hard to maintain. Constant refactoring (see section 2.6.1) and continuous changes to evolve the design to fit new needs are necessary to ensure that the characteristics of well-designed software are present.

2.3 Design Patterns

Design patterns are general solutions to problems that can commonly be found in the context of software engineering. They are reusable and therefore not application-specific. Design patterns are usually defined in terms of the structure of interfaces and classes, and their interactions.

The use of patterns in object-oriented programs has many advantages. First and foremost, when used appropriately, they provide modular and well-known solutions to problems — programmers faced with a problem for which a pattern exists do not have to reinvent the wheel. A side effect of this is that patterns are well-understood within the software engineering community and allow other developers to understand the code more quickly. This is of course of vital importance nowadays when software systems are passed onto different teams who need to learn about the architecture and design of a system in a short amount of time. The use of commonly-known patterns helps with this goal.

Even though not strictly invented by them, design patterns were popularised by the so-called “Gang of Four” (Erich Gamma, Richard Helm, Ralph Johnson, Josh Vlissides) in their book *Design Patterns: Elements of Reusable Object-oriented Software* [7] in 1995. In the book the authors describe three categories of design patterns for object oriented programs:

Creational Patterns Patterns that facilitate the creation, initialisation and configuration of objects and classes.

Structural Patterns Patterns that “are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations” [7].

Behavioural Patterns Patterns that “are concerned with algorithms and the assignment of responsibilities between classes” [7].

In the following section we discuss an example of a creational pattern. Examples for the other categories can be found in the literature [7].

2.3.1 Singleton Pattern as an Example of a Creational Pattern

One of the most widely-known design patterns that falls into the category of creational patterns is the *Singleton* pattern. In essence, it provides a way to restrict the number of instantiated objects of a class to one.

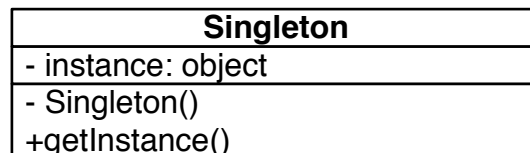


Figure 2.2: UML for singleton pattern

Figure 2.2 shows the UML diagram for the singleton pattern. This pattern is evidently one of the simplest patterns to describe: a class implementing it contains a private `instance` variable containing an instance of the class and provides a public `getInstance()` method to clients for the retrieval of that instance. Notice also that the constructor is declared `private`, thus preventing other objects from creating more instances of the class. The singleton pattern can therefore be seen as a way way to ensure that only one instance of a class is ever used in the program.

Standard UML Implementation

Following the UML diagram in figure 2.2, a possible implementation of the singleton pattern is shown in listing 2.1. Objects that choose to use the one and only `AppSettings` instance can retrieve it by calling `AppSettings.getInstance()`.

```
1 class AppSettings {
2     private static final AppSettings INSTANCE = new AppSettings();
3
4     private AppSettings() { }
5
6     public static AppSettings getInstance() {
7         return INSTANCE;
8     }
9
10    ...
11 }
```

Listing 2.1: Singleton pattern example

Despite its simplicity in the UML diagram, alternative implementations for this pattern exist.

Non-getInstance() Implementations

Although the UML diagram prescribes the use of the name `getInstance()` for the instance retrieval method, this name is not always used in practice. Instead, many implementations use contextual names.

For example, Java’s own `java.lang.Runtime` class implements the singleton pattern. Clients can retrieve a `Runtime` instance by calling `Runtime.getRuntime()` instead of `getInstance()`.

Public Static Field Implementation

In his book *Effective Java*, Bloch [8] describes a slight variation on the direct translation of the UML diagram. Instead of using a `private static` instance field to store the `AppSettings` instance and then retrieve it via a `getInstance()` method, he proposes the use of a `public static` instance field that can be accessed directly, eliminating the need for a `getInstance()` method.

Enum Implementation

Bloch also describes a second and more interesting alternative Java implementation using `enum` types. This implementation is available since Java 1.5. A sample implementation can be seen in listing 2.2.

```
1 enum AppSettings {
2     INSTANCE;
3
4     // methods go here
5     ...
6 }
```

Listing 2.2: Enum implementation of the singleton pattern

Instead of having to declare static fields and instance retrieval methods, using an `enum` type achieves the same effect in a more concise manner. Additionally, according to Bloch, this implementation also “provides the [object] serialisation machinery for free, and provides an ironclad guarantee against multiple instantiation, even in the face of sophisticated serialisation or reflection attacks” [8]. According to the author, this makes it the “best way to implement a singleton” [8].

Initialisation-on-demand Holder Idiom

Another implementation of the singleton pattern is the use of a `static` inner holder class (see listing 2.3) that contains the singleton instance.

```
1 class AppSettings {
2     private AppSettings() { }
3
4     public static AppSettings getInstance() {
5         return SingletonHolder.INSTANCE;
6     }
7
8     private static class SingletonHolder {
9         private static final AppSettings INSTANCE = new AppSettings();
10    }
11
12    ...
13 }
```

Listing 2.3: Initialisation-on-demand holder idiom

This approach loads the singleton instance lazily on the first call to `getInstance()`, as the static class is loaded on demand in the JVM. Since this implementation is thread-safe without

the need for language constructs such as `volatile` or `synchronized` [9], it is a very popular way to implement the pattern.

Final Remarks

Despite its usefulness in preventing the instantiation of multiple instances of a class, the singleton pattern is also a prime example of a pattern that is frequently overused and can actually create more damage than improve the design of the application in which it is used. One problem with using the singleton pattern that becomes apparent immediately is that if `AppSettings` contains state, using it in multiple places in the application makes the program susceptible to a wide range of bugs due to state inconsistencies, which are often found in concurrent execution contexts.

In section 2.4.2 we discuss a further usage pattern of the singleton which is problematic in the context of software design.

2.4 Code Smells

Both seasoned programmers and novices will be aware that in some cases code “just does not seem right”. This feeling is often not grounded in a specific problem at the time of discovery and it is hard to pinpoint what exactly is wrong with the code at hand. However, programmers frequently find that later on in the development process, the piece of code that seemed concerning previously now contributes to a design issue in some way, or may even be the root cause. These problems are not necessarily bugs in the software but appear to obstruct the extension or refactoring of other, possibly unrelated, parts of the code.

One of the major challenges in the field of software engineering design and refactoring is to find these problem areas accurately and consistently. However, this is a difficult task as opinions on the matter vary and the identification of problem areas in the code often “appeal[s] to some vague notion of programming aesthetics” [10]. A more solid indication as to when refactoring pieces of code may be useful is therefore needed.

In their book *Refactoring: Improving the Design of Existing Code* [10], Fowler and Beck include a chapter in which they describe the notion of “Bad Smells” in order to tackle the issue of vagueness when describing design problems. Even though the authors are unable to present precise criteria for when refactoring is overdue, they nevertheless offer a set of indications that there may be a problem in the design of the code. Notice that if one of the described indicators can be found in the source code, it does not necessarily mean that refactoring is strictly required. According to Fowler and Beck, it is advisable to pay more attention to these parts as they may be problematic. The authors’ suggestion is that observation and early refactoring may eradicate these problems.

In the following sections, we will describe some of the discussed “bad smells” and give examples to illustrate how the concept of “bad smells” can help identify potential problems in code.

2.4.1 Method Call Chains and Train Wrecks

Method call chains, also commonly known as “message chains”, are chains of subsequent method calls. For example, `foo().bar().baz()` is a method call chain, as method `bar()` is called on the result of `foo()`, and `baz()` is called on the result returned by `bar()`. Different kinds of method call chains exist, some of which can be considered “smelly code”.

Listing 2.4 shows code that contains another example of a method call chain.

```
1 class OrderDisplay {
2   public void prepare() {
3     Customer customer = new Customer();
4     ZipCode zip = customer.getAddress().getZipCode();
5     Label label = new Label();
6     label.addLine(zip.toString());
7     ...
8   }
9 }
```

Listing 2.4: Bad example of message chaining (“Train Wreck”)

The example shows the retrieval of the zip code for a given customer. However, from a purely architectural point of view, the customer does not directly have an associated zip code but instead computes it by asking its address, which can find out about the zip code. Long chains of message calls such as this result in the client (`OrderDisplay`) navigating the object structure of the code — as Fowler and Beck point out “the client is coupled to the structure of the navigation” [10]. This problem is also colloquially known as a “train wreck” [11].

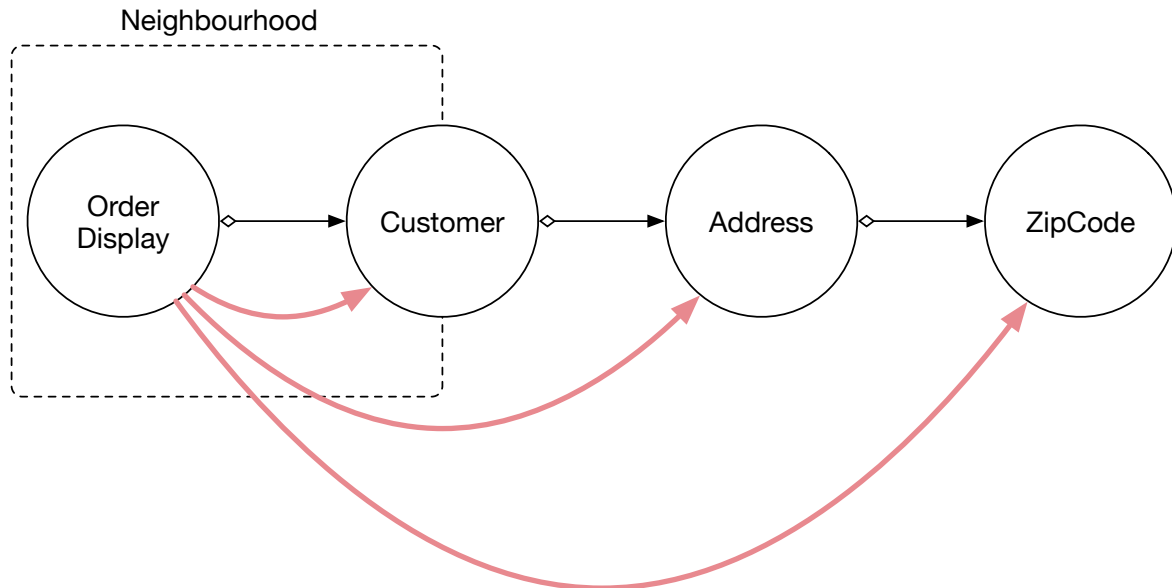


Figure 2.3: Structure of the navigation of simple train wreck example

To illustrate what is meant by the structure of the navigation, consider figure 2.3. In the diagram, object associations are drawn in thin black arrows while method calls between `OrderDisplay` and other objects are drawn in thick red lines¹.

As we can see, a `ZipCode` is associated with an `Address`, which is associated with a `Customer`, which, in turn, is owned by the `OrderDisplay` object. Examination of the method call relationships, however, paints a different picture. As it turns out, `OrderDisplay` asks all of the objects for data, even though it is only directly associated with one of them (`Customer`). This method call dependence means that there exists strong coupling between `OrderDisplay` and all other objects that are involved, whereas one would only reasonably expect `OrderDisplay` to be coupled to the neighbouring `Customer` object.

This dependence on the structure of the navigation becomes a particular problem when

¹Other method calls exist but have been left out of the diagram for clarity.

changes are made to any of the classes that are involved in this method chain. For example, in a large software project, a developer may wish to change the software to use the simpler `String` type to describe post codes instead of the custom `ZipCode` class. In this example, this change will involve changing the return type of `getZipCode()` from `ZipCode` to `String`.

```
1 class OrderDisplay {
2     public void prepare() {
3         Customer customer = new Customer();
4         ZipCode zip = customer.getAddress().getZipCode();
5         Label label = new Label();
6         label.addLine(zip.toString());
7         ...
8     }
9 }
10
11 class Customer {
12     private Address address = ...
13     ...
14
15     private doSomethingWithAddress() {
16         String zip = address.getZipCode();
17     }
18 }
19
20
21 class Address {
22     public String getZipCode() {
23         ...
24     }
25 }
```

Listing 2.5: Train wrecks make software fragile

The developer may expect to have to modify any classes that directly reference `Address`. These are classes that aggregate `Address` and therefore may call its methods. As we can see in listing 2.5, `Customer`'s `doSomethingWithAddress()` method was rightfully updated to use the new type, since `Customer` aggregates `Address`. However, since `OrderDisplay` does not aggregate `Address` in any way, the developer may assume that this class has no relation to `ZipCode`. In fact, although the number of classes is so small in this example, in real applications many thousands of classes may exist that do not aggregate `Address` and are therefore assumed to make no references to `getZipCode`. As we can see though, in the presence of train wrecks, this assumption is clearly wrong, causing compilation to fail in the current state because the developer has not made changes to the unrelated `OrderDisplay` class.

This example shows the fragility of software designs that feature train wrecks; a simple change in one class may affect classes that are seemingly unrelated, making it not only difficult to maintain and improve the existing code, but also to reuse it elsewhere.

Lieberherr, Holland and Riel further discuss the issues associated with these kind of message chains further in their paper “Object-oriented programming: an objective sense of style” and also introduce the *Law of Demeter* in an attempt to give a guide on how to avoid these long method chains that involve asking for data and ultimately improve code quality [12]:

For all classes `C`, and for all methods `M` attached to `C`, all objects to which `M` sends a message must be instances of classes associated with the following classes:

1. The argument classes of `M` (including `C`).
2. The instance variable classes of `C`.

In other words, methods in a class A should only communicate with immediate neighbours of class A. In addition, by interpreting communication between objects as sending messages, rather than asking for data, indicates that objects should *tell* their neighbours what to do, not *ask* them for information. This results in the commonly known piece of advice “Tell, don’t ask” [13] coined by Hunt and Thomas.

Referring back to the example in figure 2.3, `OrderDisplay` should therefore only interact directly with its neighbouring `Customer` object. To achieve this in our example, the `OrderDisplay` could *tell* the `Customer` to fill the label with the correct information as shown in listing 2.6.

```

1 class OrderDisplay {
2     public void prepare() {
3         Customer customer = new Customer();
4         Label label = new Label();
5         customer.fillLabel(label);
6         ...
7     }
8 }

```

Listing 2.6: Avoid train wrecks by limiting interaction to neighbouring objects

The associated object interaction diagram in 2.4 clearly shows the affect of the changes: only neighbouring objects communicate with each other. Making changes to `Address` is now easier as the effects of the changes would only affect objects associated with `Address`.

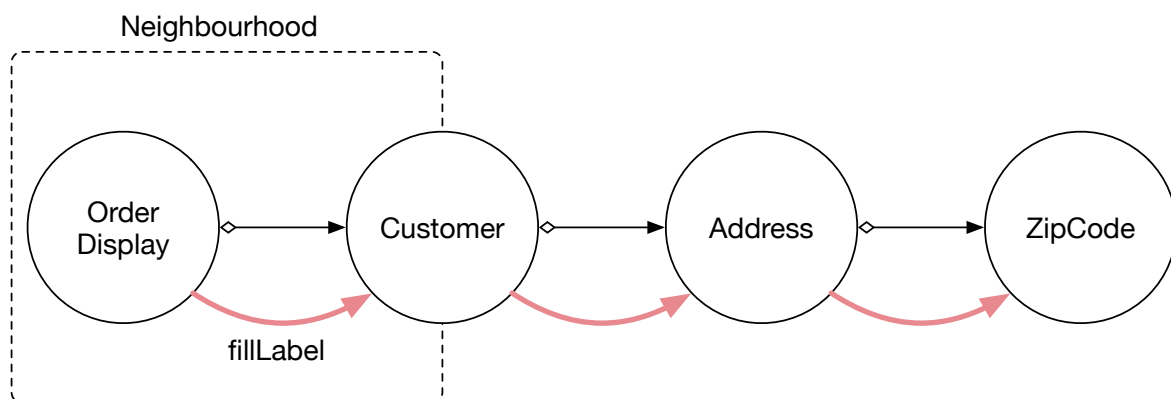


Figure 2.4: No more train wrecks: objects only *tell* neighbours to perform tasks

In the absence of train wrecks, making changes to the classes is now easier. Consider again the previous example in which a developer wishes to migrate to the usage of `String` instead of `ZipCode`.

Note first of all that the `OrderDisplay` class in listing 2.6 no longer even makes references to the concept of a zip code. The code instead more abstractly describes the task of filling a label.

Listing 2.7 shows that, as before, the `Address` and `Customer` classes were modified. Since `Customer` aggregates `Address`, this is to be expected. On the other hand, `OrderDisplay` did not need to be changed, showing that the new design is less fragile.

Note that one may wish to further refactor this code to delegate the filling of the label to the `Address` instead of performing the task inside the `Customer`!

```

1 class OrderDisplay {
2     public void prepare() {
3         Customer customer = new Customer();
4         Label label = new Label();
5         customer.fillLabel(label);
6         ...
7     }
8 }
9
10 class Customer {
11     ...
12
13     private fillLabel(Label label) {
14         String zip = address.getZipCode();
15         label.addLine(zip);
16     }
17
18     private doSomethingWithAddress() {
19         String zip = address.getZipCode();
20     }
21 }
22
23 class Address {
24     public String getZipCode() {
25         ...
26     }
27 }

```

Listing 2.7: Changing types is easier without train wrecks

Fluent Interfaces in the Context of Method Chains

Although many method call chains fit the description of a train wreck, there are some chains of method calls that do not suffer from being reliant on knowledge about deeply nested object structures.

For example, consider the example code given in listing 2.8. In this example the “builder pattern” is used to construct a `Request` instance: instead of passing the arguments for the url, content and number of allowed send attempts of the request to its constructor directly, this code uses a builder which allows the client to name each parameter. In practice, this pattern is often used to construct complex objects that have unmodifiable state.

```

1 public class BuilderExample {
2     public void execute() {
3         Request.Builder b = new Request.Builder();
4         Request request = b.withUrl("http://example.com").withContent("")
5                             .withAttempts(10).build();
6     }
7 }
8
9 class Request {
10     ...
11     static class Builder {
12         ...
13         public Builder withUrl(String url) {
14             this.url = url;
15             return this;
16         }
17         ...
18     }
19     ...
20 }

```

Listing 2.8: Method chain conforming to law of demeter

The implementation of this builder pattern is a good example of a successful application of a “fluent interface” [14]. A fluent interface is one that allows developers to write code against said interface *fluently* — often this is realised by being able to chain method calls to form English-like structures that are easy to read and yet compute a desired result, e.g. `TimeInterval meetingTime = fiveOClock.until(sixOClock);`. The request builder in listing 2.8 is implemented in a similar way that allows clients to chain `with*` calls to form an easily-readable construction of a `Request`.

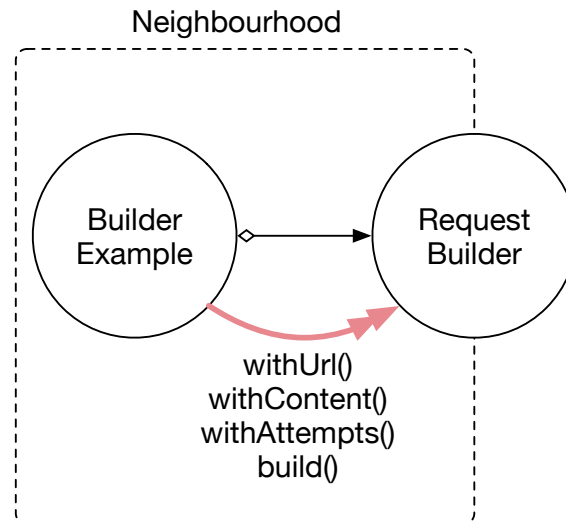


Figure 2.5: Object navigation structure of builder pattern method chain

Figure 2.5 shows the object navigation structure of the application of the builder pattern. As we can see, `BuilderExample` only interacts with one object: an instance of `Request.Builder`. This means that despite featuring a method chain, the client code in `BuilderExample` only communicates with its immediate neighbours — this code does not suffer from the train wreck problem.

Lieberherr and Holland and Riel’s description of the train wreck problem takes this consideration into account and allows the use of fluent interfaces — the example in listing 2.8 is therefore not a train wreck.

2.4.2 Direct Use of Singleton

Naturally, not all code that should be refactored is covered by one or more code smells described in *Refactoring: Improving the Design of Existing Code* [10]. An example of code that seems “smelly” despite not appearing in the book is the “direct use of a singleton”, best explained through an example. Consider the code in listing 2.9.

The code implements a simple `Mailer` class that provides a method to send emails. In this example, all emails’ `FROM:` fields are always set to the same address, one that is specified in the settings of the application (possibly in a configuration file). To retrieve this address the `Mailer` uses `AppSettings` to read the value.

`AppSettings` uses the singleton pattern (see section 2.3.1) which means that the one and only instance of this class can be retrieved via a call to the `static getInstance()` method. Notice that `AppSettings.getInstance()` can be called from anywhere in the application to retrieve said instance. One example of an instance retrieval can be found in the `Mailer`’s `send()` method: `AppSettings.getInstance().getMailFromAddress()`.

```

1 class Mailer {
2     public void send(String address, String body) {
3         String fromAddress = AppSettings.getInstance().getMailFromAddress();
4         ...
5     }
6 }
7
8 public class AppSettings {
9     private static final AppSettings INSTANCE = new AppSettings();
10
11     private AppSettings() { }
12
13     public static AppSettings getInstance() {
14         return INSTANCE;
15     }
16
17     String getMailFromAddress() { ... }
18 }

```

Listing 2.9: Direct use of singleton in mailer

With this in mind, consider the relationship between `Mailer` and `AppSettings`. Evidently, the `Mailer` implementation is coupled to the `AppSettings` class. However, even worse, it is also tied to one specific *instance* of the `AppSettings` class. In practice, this means that the `Mailer` can never be used without the singleton `AppSettings` instance. While this may seem acceptable in a normal production environment, it introduces problems in other scenarios, first and foremost when testing the application. In a setting in which `AppSettings` reads data from a file system or works with real production data, it is impossible to test the `Mailer` class in an isolated test environment. Further, since `Mailer` is tied to this application's `AppSettings`, it is also difficult to reuse the `Mailer` in another module or application because of its dependence on the singleton.

Interestingly, all of these problems were introduced by only one call to `AppSettings.getInstance()`, a call that is available throughout the entire project — as a developer it is very easy to call this method in any class in the project if one requires to read the settings of an app. By doing so in many places, one misuses the singleton pattern by treating `AppSettings` as a global variable, inadvertently tying many parts of the application to this one specific instance, increasing coupling throughout the application and making further development more difficult.

Consider now an alternative usage pattern of the singleton in the revised and refactored version of the same code in listing 2.10.

In this revised version, we were able to reduce the coupling between the classes by first introducing a `Settings` interface that abstracts the capabilities of `AppSettings` and then injecting an instance into the `Mailer` class. This is also called dependency injection as `Mailer`'s dependencies are injected into the class on construction. The client no longer requires knowledge about how many settings instances may exist and where settings initialised. Instead, it expects a conforming `Settings` instance to be passed in for use.

Listing 2.11 illustrates how different kinds of `Settings` can now be constructed that are compatible with the `Mailer`. This functionality can be useful when unit testing the application.

However, even outside of the context of testing, it is now easier to reuse the components of the application. Since `Mailer` is no longer tied to the application's `AppSettings` but instead only requires a object that conforms to the `Settings` interface, it is easy to use `Mailer` in other applications.

```

1 interface Settings {
2     public String getMailFromAddress();
3 }
4
5 class Mailer {
6     private Settings settings;
7
8     public Mailer(Settings settings) {
9         this.settings = settings;
10    }
11
12    public void send(String address, String body) {
13        String fromAddress = settings.getMailFromAddress();
14        ...
15    }
16 }
17
18 class AppSettings implements Settings {
19     private static final AppSettings INSTANCE = new AppSettings();
20
21     private AppSettings() { }
22
23     public static AppSettings getInstance() {
24         return INSTANCE;
25     }
26
27     String getMailFromAddress() { ... }
28 }

```

Listing 2.10: Dependency injection of settings

```

1 class FakeSettings implements Settings {
2     public String getMailFromAddress() {
3         return "test@example.com";
4     }
5 }
6
7 ...
8 Mailer fakeMailer = new Mailer(new FakeSettings());
9 Mailer realMailer = new Mailer(AppSettings.getInstance());
10 ...

```

Listing 2.11: Different kinds of settings can be injected

2.4.3 Long Method

According to Fowler and Beck, one of the most commonly occurring “bad smells” in code are “long methods” which, according to the authors, often indicate a problem in the broader object-oriented design.

Despite its name, the “long method” smell does not actually describe the literal length (number of lines of code [LOC]) of a method but rather the “semantic distance between what the method does and how it does it” [10]. A large semantic distance makes the method more difficult to understand for a programmer reading the code.

This semantic distance can be introduced by a large number of statements (i.e. high LOC) but is also often found when a method does too many things. Fowler and Beck therefore state that a good indication for high semantic distance is the need to comment parts of a method and suggest that for every comment describing a few lines of code or even just a single line, a new method with a fitting name should be created instead.

Aside from comments, according to Fowler and Beck, loops and `if`-statements are also often a sign for code that should be extracted into its own method.

Consider the example in listing 2.12 taken [15].

```
1 protected String rtrim(String s) {
2     // if the string is empty, do nothing and return it
3     if ((s == null) || (s.length() == 0)) {
4         return s;
5     }
6
7     // get the position of the last character in the string
8     int pos = s.length();
9     while((pos > 0) && Character.isWhitespace(s.charAt(pos - 1))) {
10        --pos;
11    }
12
13    // remove everything after the last character
14    return s.substring(0, pos);
15 }
```

Listing 2.12: Long Method Example

As we can see, the effective lines of code in this method is 8, a relatively low number which would, on its own, not indicate any problems. However, one may argue that something still feels wrong about this method. Indeed, the attributes of a “long method” according to Fowler and Beck can be found:

- Three comments before the start of different blocks of code suggest separate parts of the code
- No immediate relationship between trimming the string and counting whitespaces in a loop can be found

In this case, we deem it appropriate to extract parts of this method into their own methods to reduce the semantic distance between what the method does and how it does it. Listing 2.13 shows how three simple extractions of code into their own methods lead to more modular and reusable code.

```
1 protected String rtrim(String s) {
2     if (isEmpty(s)) {
3         return s;
4     } else {
5         int cutPos = getLastCharacterPosition(s);
6         return removeAfterPosition(s, cutPos);
7     }
8 }
9
10 private boolean isEmpty(String s) {
11     return (s == null) || (s.length() == 0);
12 }
13
14 private int getLastCharacterPosition(s) {
15     int pos = s.length();
16     while((pos > 0) && Character.isWhitespace(s.charAt(pos - 1))) {
17         --pos;
18     }
19     return pos;
20 }
21
22 private String removeAfterPosition(String str, int cutPos) {
23     return str.substring(0, cutPos);
24 }
```

Listing 2.13: Refactored code no longer suffers *Long Method* smell

The new code also serves as an example to illustrate the benefits of very small methods with sometimes as few as one line of code. Rather than only being able to `rtrim()` a string, the refactored code allows software engineers to reuse a lot of the functionality that makes up the `rtrim()` method.

Furthermore, the new code is self-documenting due to descriptive names of methods that only perform one specific task. Even though the overall effective number of lines of code has increased, the code no longer appears smelly — it seems well-modularised, illustrating that constant refactoring and small methods effectively extend the software engineer’s tool belt and is therefore advisable.

2.5 Learning About Software Design

Software design is an essential skill as a software engineer. It is therefore no surprise that there are many different ways in which engineers can learn about the topic:

- Personal experience
- Mentoring by more experienced engineers
- Books and other publications
- Meetups and Discussions
- Courses (e.g. “Software Engineering Design” courses at university)

Personal experience is generally considered the most valuable way to learn about software design, because “good design comes from experience, and experience comes from bad design” [16]. However, this approach inherently suffers from an extremely long feedback loop as problems are usually only realised after a long time. As discussed in section 2.1, fast feedback loops are preferable as they allow for fast adjustments and easier learning.

In the context of feedback loops, being mentored by more experienced engineers is very effective. A mentor is able to give quick feedback and can answer many of the questions that a new programmer may have using his or her experience. This is particularly the case when using techniques such as pair-programming, during which the mentor can instantly give feedback and suggest design changes.

The other approaches mentioned above are built on the concept of learning from other developers’ experiences. While this is certainly useful, it can be difficult for new software engineers to relate the experiences described in books, discussions and course materials to their own software projects. With a lack of prior experience, it takes a long time to fully understand the consequences of a particular design decision in one’s own software. As with personal experience, feedback loops are therefore generally slow. Bad design decisions cannot be identified immediately by an external entity (such as a mentor) who is more familiar with the advantages and disadvantages of the particular design pattern.

Teaching Design Patterns

One example of how the lack of a fast feedback loop can impede the learning outcome is given by Chatzigeorgiou, Tsantalis, and Deligiannis in their paper “An empirical study on students’ ability to comprehend design patterns.” The authors describe an experiment in which they asked students to submit two projects with the same functionality, one with and one without

the use of design patterns. The students were asked to evaluate their designs using metrics and explain their reasoning for each design pattern they used.

Chatzigeorgiou, Tsantalis, and Deligiannis report that students had difficulty relating changes in metrics back to the introduction of specific patterns. Although students used design patterns, they “had difficulties in relating design patterns to specific problems they solve” [17]. These findings support the view of Astrachan, Mitchener, Berry, *et al.* that “the strength, purpose, and abstractness of design patterns makes them very accessible to those well-versed in object technology, but less so to those new to the field” [16].

Relating Personal Experience to Teaching Software Design

These results indicate that teaching both design patterns and software design abstractly, is not enough for students to be able to successfully integrate what they have learnt in their own code. We therefore argue that a continuous and fast feedback loop that informs developers about potential design flaws in their code as they appear would be beneficial to software design teaching. Instead of working with abstract example programs, *Vignelli* suggests possible design improvements in real-world applications, that the developer is currently working on. We believe that this will help in the transition to object-oriented thinking.

2.6 Development Techniques

2.6.1 Refactoring

One of the most prominent techniques to improve code quality is the act of refactoring. Martin Fowler defines refactoring as “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour” [10, p. 53]. Refactoring is therefore a technique that allows software engineers to develop their design “after [the code] has been written” [10]. This makes refactoring a core practice of modern development processes that are often designed around tight feedback loops and iteration; iterating on the design, rather than building it all up front, allows software to grow and develop organically.

The main reasons to refactor code are to make it more understandable and more maintainable. It is for this reason that authors such as Fowler and Beck recommend refactoring *all the time* instead of setting aside time to refactor code. In the words of Fowler, this is because “you refactor because you want to do something else, and refactoring helps you do that other thing” [10].

Since refactoring necessarily involves changing the structure of the code, testing is extremely important. There is a risk involved in making changes to working code if one cannot be sure that the resulting code will still function correctly. Automated tests help alleviate that problem: a well-tested program can be transformed in such a way that the programmer has great certainty that the restructuring of the code has not altered the observable behaviour. Techniques such as test-driven development incorporate refactoring into their tight feedback loops. In his book *Refactoring to Patterns*, Kerievsky describes how this application of “continuous refactoring” helps keep the overall quality of a project high, even though he admits that it can take time to get used to this process [18, pp. 4–6].

Refactoring To Patterns

Refactoring can be understood as transforming a software from an original design to another. It is therefore evident that code smells (see section 2.4), the act of refactoring, and design patterns (see section 2.3) are closely related. In fact, in *Design Patterns: Elements of Reusable Object-oriented Software* Gamma, Helm, Johnson, *et al.* note that “design patterns capture many of the structures that result from refactoring. [...] Design patterns thus provide targets for your refactorings” [7, p. 354].

In his book *Refactoring to Patterns* Kerievsky explains this relationship further by relating commonly found anti-patterns to better design patterns via a set of refactoring steps. These composite refactorings (refactorings consisting of multiple steps) walk the reader through the process of refactoring code suffering from a particular problem to code that solves this problem using a design pattern. Note that what Kerievsky promotes is refactoring to patterns to *solve a problem*, rather than to find excuses to use a particular pattern for the sake of it being a pattern.

The author goes on to list a number of refactoring strategies to solve particular problems and gives motivations for each problem. For example, one of the refactorings that are described is the “Inline Singleton” refactoring. In this chapter, Kerievsky explains that if “your code needs access to an object but doesn’t need a global point of access to it”, one should “move the singleton’s features to a class that stores and provides access to the object [and] delete the singleton” [18, p. 114], thus linking a smell (“global data access”) to a solution by way of refactoring.

After explaining the motivation further and listing advantages as well as disadvantages of the refactoring, the chapter outlines a composite refactoring consisting of three actionable steps which are given here as an example: [18, pp. 117–118]

1. “Declare the Singleton’s public methods on your absorbing class. Make the new methods delegate back to the Singleton, and remove any ‘static’ designations they may have (in the absorbing class). [...]”
2. “Change all the client code references to the Singleton to references to the absorbing class.”
3. “Use *Move Method* [F] and *Move Field* [F] to move features from the Singleton to the absorbing class until there is nothing left. [...]”
4. “Delete the Singleton.”

These steps are a good example of how a developer may move from a code smell towards a better design via a small number of pre-determined refactoring steps. *Vignelli* codifies composite refactorings of this form and guides the programmer through them in the context of their own code.

2.6.2 Manual Code Review

One of the oldest, yet most effective, techniques for improving code quality is manual code review. During manual code review, software engineers who have completed a feature ask one or more team members to review the code they have written. Although time-consuming, manual code review is widely used in industry. This is not only due to its effectiveness in maintaining high quality standards but also helps spread knowledge about other components in the software.

In *Refactoring: Improving the Design of Existing Code*, Fowler and Beck note the apparent correlation between a programmer’s ability to find code smells intuitively and their experience

— more experienced engineers tend to see problems more quickly and reliably [10]. Manual code review exploits this insight by promoting collaboration between many programmers who all have different experiences. With the combined knowledge of all participating team members, the likelihood for a problem (functional or design) to be found and corrected increases.

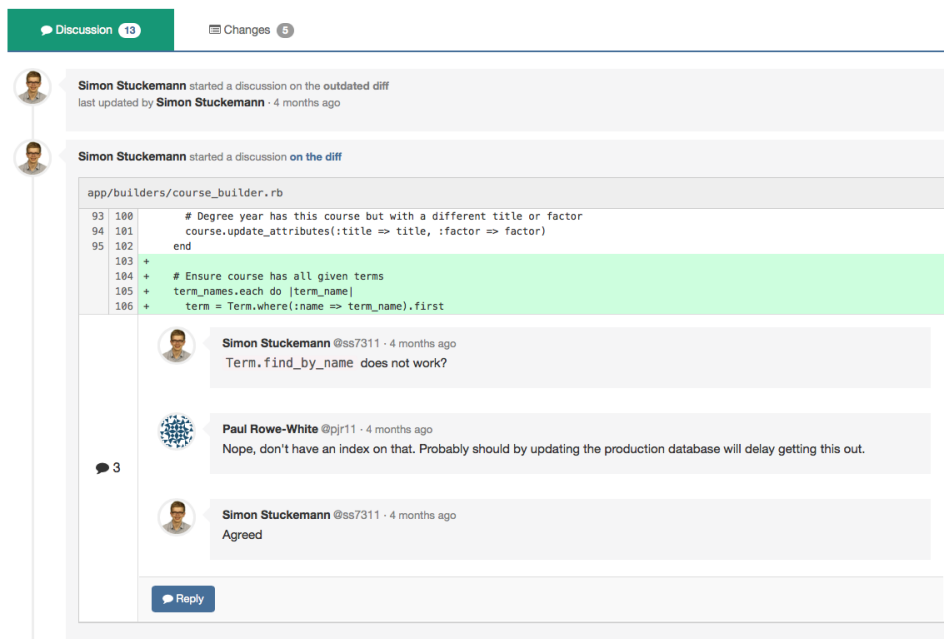


Figure 2.6: Manual code review on gitlab

Figure 2.6 shows an example of a discussion during manual code review. As we can see, reviewers can ask questions which can lead to interesting discussions, ultimately resulting in insights for further work that needs to be done. This feedback loop is one of the major advantages of the manual code review process: participants are able to learn from each other, while at the same time ensuring that code changes do not negatively impact the functionality or design of the product.

Arguably, many modern version control tools such as Github² or Gitlab³ do not offer appropriate representations of the changes that allow programmers to identify design problems. This is because current systems often only show the latest additions and deletions to and from a file, thus highlighting individual lines (see green lines in figure 2.6) but not giving an overview of the current system design. This makes it difficult for reviewers to give feedback on how the new code fits into the old design and whether any changes should be made.

Another problem with manual code review is the lack of consistency in the reviews. Due to different previous experiences in the software design field, feedback from co-workers often varies significantly due to the subjectivity that is involved in the process. There is no single best solution and multiple approaches may have their merits.

Lastly, manual code review also suffers from comparatively long feedback loops. Even though code is reviewed by team members who can comment on the style and point out potential problems, this step is often only done just before the planned release of the new feature. This makes the suggestion of design changes difficult, especially if reviews generally take a long time. Even when reviews only take one hour or less, programmers are unable to get feedback for the code they write as they are writing it, using traditional code reviews. At the point of code review, it can be difficult to change aspects of the system substantially without considerable

²<https://github.com>

³<https://gitlab.com>

time and cost consequences. This means that bad design decisions are more likely to find their way into the software.

2.6.3 Testing

One of the most popular processes for improving code quality is the act of testing. Testing can take multiple forms, including unit testing and integration testing. Unit tests check individual components of the system for bugs in a controlled environment, while integration tests aim to test the interaction between multiple components of the system.

Most testing practices aim to reduce the number of bugs in the systems under test. These functional tests are important, but rarely have considerable influence in the design of the system. On the other hand, some approaches also aim to improve code quality overall, which includes system design.

One such approach is the use of “mock objects”, first described by Tim Mackinnon in 2001 [19]. Mock objects allow engineers to think about the interactions between objects instead of changes to objects’ state when writing tests. This is especially helpful when testing classes that are still under development and whose dependencies may not yet have been created.

The benefits of mock objects were discovered in the light of test-driven development practices. Test-driven development (TDD) is a development process that relies on developers first writing a test for a new feature/addition, then writing the minimal code required to make the test pass, refactoring the resulting code, and finally repeating the process.

Many authors today agree that the use of test-driven development with mock objects leads to better-designed software that is more modular [19]–[21], which makes components easier to reuse.

One of the major advantages of this technique is therefore the improvement of code design through testing, a part of the development process that should be performed in any software development project. However, it is also worth noting that approaches to testing vary significantly in different organisations and that TDD is only very rarely employed fully in the development process [22]. In addition, although practicing TDD often leads to better software design, the use of the technique does not guarantee that developers learn more about different design approaches and potential problems with some design decisions. This means that one cannot rely solely on testing practices to enforce good design.

2.7 Tools

2.7.1 Integrated Development Environments (IDEs)

An Integrated Development Environment (IDE) is an application aimed at helping software engineers develop their products by integrating some of the most common components that are usually needed during the development process. They usually consist of a source code editor, a debugger, and software to automate builds. Modern IDEs also commonly feature code completion and tools that help during the refactoring process (see section 2.6.1).

According to Wikipedia, the “the boundary between an integrated development environment and other parts of the broader software development environment is not well-defined” [23]. Many IDEs therefore offer plugin systems whereby other developers can create extensions and provide additional functionality, such as interfaces for source control, file tree browsers, or even easy-to-access status reports for automated builds.

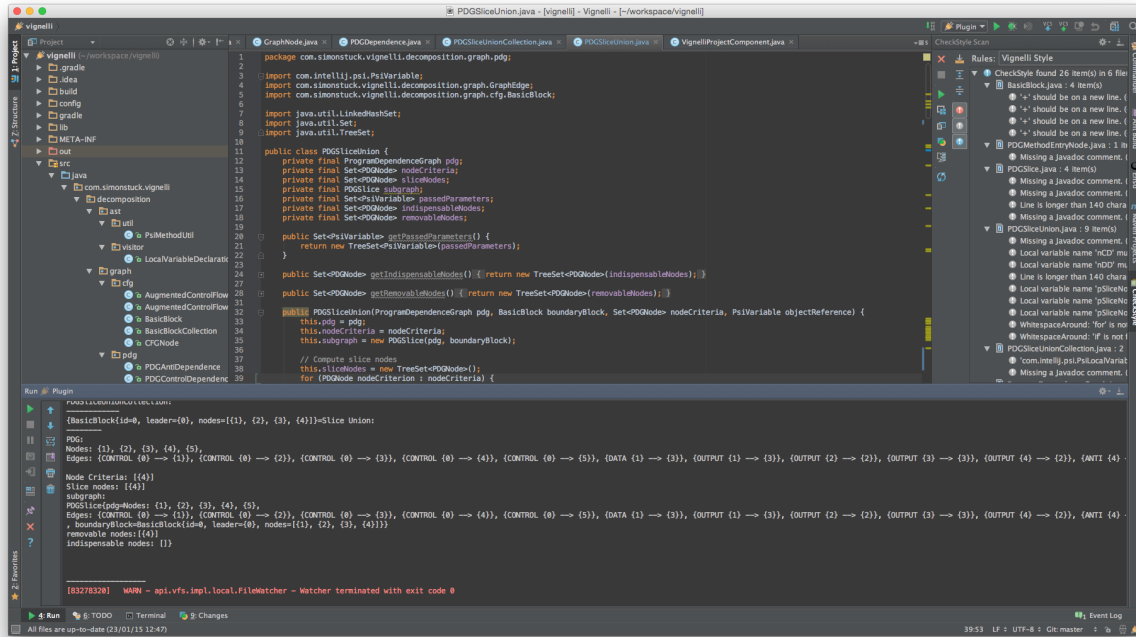


Figure 2.7: IntelliJ IDEA integrated development environment

Figure 2.7 shows a screenshot of *IntelliJ IDEA*, one of the most popular IDEs that is currently used extensively in industry [24]. Although IntelliJ was primarily built for use with Java and thus offers vast support for many of the language’s features, users are able to install language plugins that allow the software to be used in other environments. For instance, IntelliJ is also one of the recommended *Scala* programming environments⁴.

Figure 2.7 shows IntelliJ being used for Java development. Packages and classes can be explored in the left pane, while debugging output is shown in the lower pane. Similarly to other IDEs on the market, IntelliJ’s source code editor features syntax highlighting and even intelligent code completion (see figure 2.8), which intelligently and automatically suggests methods that the developer may wish to call on an object, which can significantly reduce development time.

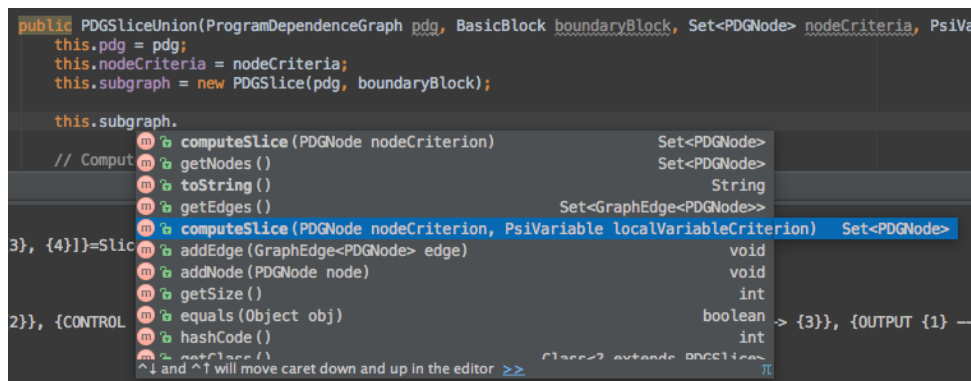


Figure 2.8: IntelliJ IDEA autocompletion

Apart from helping with the production of code, modern IDEs such as IntelliJ can also help identify problems in the source code that may not be immediately obvious to the programmer.

⁴The *Scala* download page suggests IntelliJ as one of the possible options <http://www.scala-lang.org/download/> (last accessed: 25. January)

Figure 2.9 shows the software suggesting a simplification of an `if`-statement.

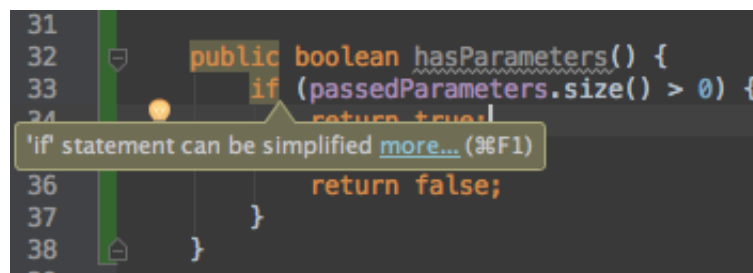


Figure 2.9: IntelliJ IDEA refactoring suggestion

Although this on-the-fly identification of problems is generally considered helpful, it is worth noting that highlighted problems are often of local nature, meaning that problems are bound to only a few lines of code contained within a relatively small block of code, typically a method. This locality results in relatively simple refactorings that can be suggested such as the one depicted in figure 2.9.

On the other hand, on-the-fly, high-level analyses and refactoring strategies such as the identification and eradication of design flaws involving multiple classes, their interactions and the program control flow, are currently not available using IDEs such as IntelliJ or its competitors (e.g. Eclipse⁵).

Nevertheless, IDEs also commonly aid engineers in performing code changes that affect large parts of the system. Some of the most common refactoring steps are packaged into IntelliJ and other IDEs. These allow programmers to extract selected code into its own method, move methods to other classes, or even move methods within the class hierarchy for generalisation. Although these changes can be performed by hand, this would typically take a long time and is also error-prone, particularly when test coverage of the affected code is low. Automated refactoring tools therefore speed up this process and make the steps less error-prone. Automated ways to refactor with little risk to break existing functionality also therefore reduce engineers' resistance to refactorings.

IDEs are all-in-one environments that make it easier for software developers to switch contexts between different parts of the development process, such as debugging. Close integration between the components (e.g. specifying breakpoints for the debugger by marking the corresponding lines of code in the source code editor) makes it easier to use multiple tools together. This “has the potential to improve overall productivity” [23] during development.

2.7.2 FindBugs

*FindBugs*⁶ is a static source code analysis tool that aims to find bugs in Java source code. Unlike other tools and processes discussed in this section, FindBugs focuses less on the identification of design flaws but instead on bug-finding. Among its many capabilities are the ability to identify:

- Possible `null` pointer dereferences
- Checking string equality with `==`
- Not overriding both `equals` and `hashCode` (a requirement in Java that is not enforced by the compiler, but, when ignored, can cause significant bugs)

⁵<https://www.eclipse.org>

⁶<http://findbugs.sourceforge.net>

FindBugs is one of the most popular pieces of software that aim to improve overall software quality. Many IDE plugins, GUIs and other interfaces exist for this tool, and are heavily used by developers in the community — its popularity shows that the developer community embraces tools that aim to improve overall software quality.

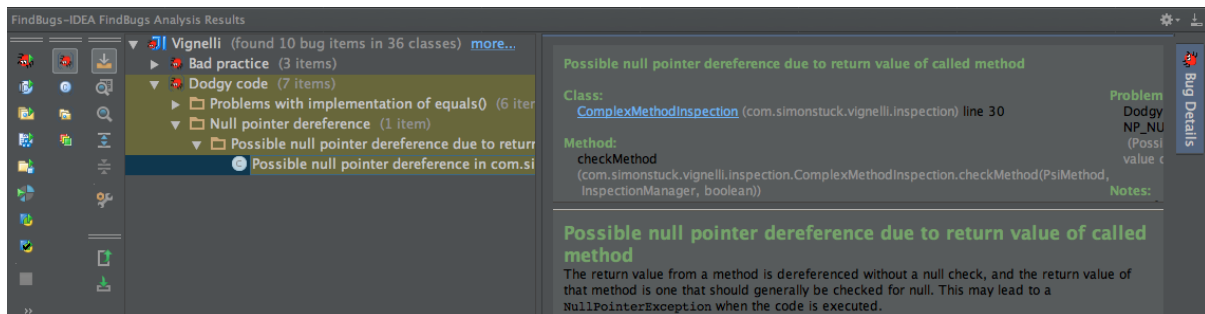


Figure 2.10: FindBugs IntelliJ plugin identifying problems in Java source code

Figure 2.10 shows the *FindBugs-IDEA*⁷ plugin interface for IntelliJ. We include the screenshot as an example of a successful interface. The panel shows both a list of problems but also gives the user more details about what could be a problem.

FindBugs is also an interesting example due to the techniques it uses to identify bugs in the code, which differ greatly from traditional formal verification and bug-finding techniques used in other products. FindBugs’ innovative and pragmatic technique is discussed further in section 2.8.2.

2.7.3 Checkstyle

According to its official website⁸, “Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task. This makes it ideal for projects that want to enforce a coding standard” [25].

Similarly to *FindBugs*, *Checkstyle* does not attempt to identify design- or architecture-level issues in the code, but instead concentrates on more tractable, local issues in the code. The tool focuses on enforcing a consistent coding style throughout the project.

To achieve this, the software is highly configurable: engineers can provide custom configurations written in XML that describe the desired style guide the software should follow. For example, for the *Vignelli* source code, we are using the tool to ensure, among other rules, that:

- All imports are fully qualified (i.e. no * imports)
- Conditional blocks use braces
- etc.

As we can see by this list, the tool focuses purely on programming style, essentially formalising style guides that are widely available and developed for multiple programming languages.

Checkstyle is a command-line tool but a number of plugins for many different development environments are available.

⁷<https://andrepdo.github.io/findbugs-idea/>, last accessed: 10 June 2015

⁸<http://checkstyle.sourceforge.net>, last accessed: 10 June 2015

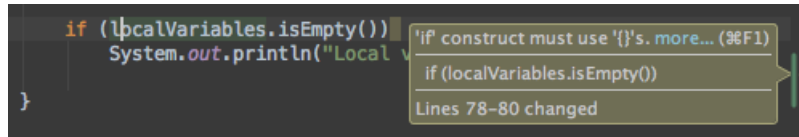


Figure 2.11: IntelliJ Checkstyle plugin integrates with built-in warning system

Figure 2.11 shows how the *Checkstyle* plugin integrates with IntelliJ’s built-in warning system and highlights parts of the code that violate the specified rules. This particular plugin also runs continuously, which means that programmers do not have to specially invoke it to find style issues in the code. The constant feedback loop that this approach delivers is beneficial to the code quality overall, as the IDE encourages engineers to fix problems directly as they are introduced.

Even though *Checkstyle* does not provide easy ways to refactor the code automatically, we find that this is not an issue in practice. Due to the nature of code style rules, which are largely focused on local structure of the code rather than higher-level organisation, fixes can be performed manually without considerable effort. This makes the IntelliJ Checkstyle plugin a prime candidate for extension with automated refactoring capabilities via IntelliJ’s “Quick fixes” which allow developers to fix issues with one-click.

2.7.4 PMD

PMD is a static source code analysis tool that is integrated in many of today’s most popular IDEs via their plugin systems. According to its official website⁹ the tool “finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth” [26].

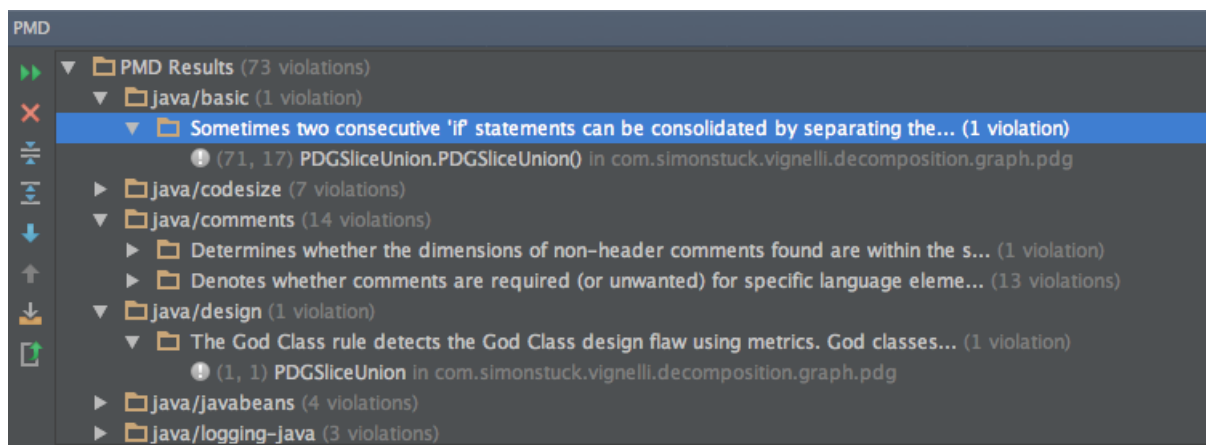


Figure 2.12: PMD IntelliJ plugin identifying problems in Java source code

As we can see in figure 2.12, the PMD tool is able to identify a large number of different code smells, covering basic improvements such as the consolidation of two consecutive `if`-statements or the identification of God classes, classes that “know too much or do too much” [27].

Serving as an analysis tool, PMD does not suggest ways to refactor the code to eliminate the problems it finds but instead expects the user to evaluate its findings and act on them. PMD is able to identify the existence of problems the code suffers from, but is unable to give more detailed explanations that link the theory to the code itself.

⁹<http://pmd.sourceforge.net>, last accessed: 10 June 2015

For example, in figure 2.12 we can see that PMD states that the `PDGSliceUnion` suffers from a “God class” design flaw. Clicking on the message only navigates to the class though, leaving the engineer to find out what parts of the class cause it to “know too much or do too much”. We believe a more useful diagnostic would also identify more specific problem areas in the class, such as suggesting methods that together are self-contained and could be extracted. Admittedly, this functionality would fall out of the realm of pure problem identification software — however, we believe that to make the tool truly useful, going the extra step is vital in helping engineers develop their design skills.

It is also worth noting that even though PMD is able to provide relatively detailed explanations as to what anti-patterns are used in the code, it is not able to do so in a continuous fashion: running the analysis requires a manual invocation and takes a few seconds to run, making it an impractical tool for constant feedback.

A further disadvantage of the IntelliJ implementation of the tool is that despite explanations of the identified issues are clear, they only appear in a tooltip for a few seconds when hovering the mouse over an issue, making it a frustrating user experience.

PMD is also able to run as a command-line tool (see figure 2.13). This makes it suitable for use in most continuous integration (CI) systems¹⁰.

```
simonstuck-pro → vignelli git:(master) ✕ pmd pmd -d src/main/java/com/simonstuck/vignelli/ -rulesets java-basic
/Users/Simon/workspace/vignelli/src/main/java/com/simonstuck/vignelli/decomposition/graph/cfg/AugmentedControlFlowGrap
hFactory.java:77: Overriding method merely calls super
/Users/Simon/workspace/vignelli/src/main/java/com/simonstuck/vignelli/decomposition/graph/pdg/PDGDependence.java:35:
  Ensure you override both equals() and hashCode()
/Users/Simon/workspace/vignelli/src/main/java/com/simonstuck/vignelli/decomposition/graph/pdg/PDGSliceUnion.java:71:
  These nested if statements could be combined
```

Figure 2.13: PMD command line tool

2.7.5 JDeodorant

Unlike the other tools and processes described in this section, *JDeodorant*¹¹ is built solely as an *Eclipse* plugin and therefore not easily portable to other programming environments such as IntelliJ or indeed non-IDE programming setups.

JDeodorant is similar to this project in that it aims to identify design problems in software and then attempts to suggest possible refactorings to the source code. It currently¹² supports the resolution of four different kinds of bad smells in code:

- Feature Envy¹³
- State Checking¹⁴
- God Class (see section 2.7.4 for explanation),
- Long Method (see section 2.4.3)

¹⁰JetBrains’ *TeamCity* CI server supports running *all* IntelliJ inspections, allowing it to use any plugins to be part of the automated build process. However, this is an exception and not commonly supported in other products.

¹¹<http://jdeodorant.com>, last accessed: 10 June 2015

¹²last checked on 10 June 2015

¹³The “Feature Envy” smell occurs when one method “seems more interested in the properties of a class other than the one it is actually in” [28]. Refer to [10] more details.

¹⁴The *state checking* smell occurs when object-oriented code varies behaviour based on class types of arguments. This can be resolved by replacing conditionals with polymorphism (see <http://www.refactoring.com/catalog/replaceConditionalWithPolymorphism.html> for more detail).

JDeodorant is a batch processing plugin which analyses the entire project code when it is invoked. Analysis of a project typically takes a few seconds after which the tool reports improvement opportunities to the developer. However, note that these are refactoring opportunities that are not necessarily computed based on whether the tool deems the code “smelly”, but instead on whether refactorings exist that are generally understood to improve the design of the code.

For example, instead of classifying methods as “long methods” or not and subsequently computing possible refactoring opportunities that may improve the design, *JDeodorant* finds a set of computation slices (e.g. the full computation of one variable inside a method) that can be extracted. These refactorings are then suggested to the user under the assumption that extracting separate computations into their own respective methods will improve the overall design by making the code more modular. As a result of this approach, the plugin explains which refactorings to apply, but not why they should be applied (see figure 2.14).

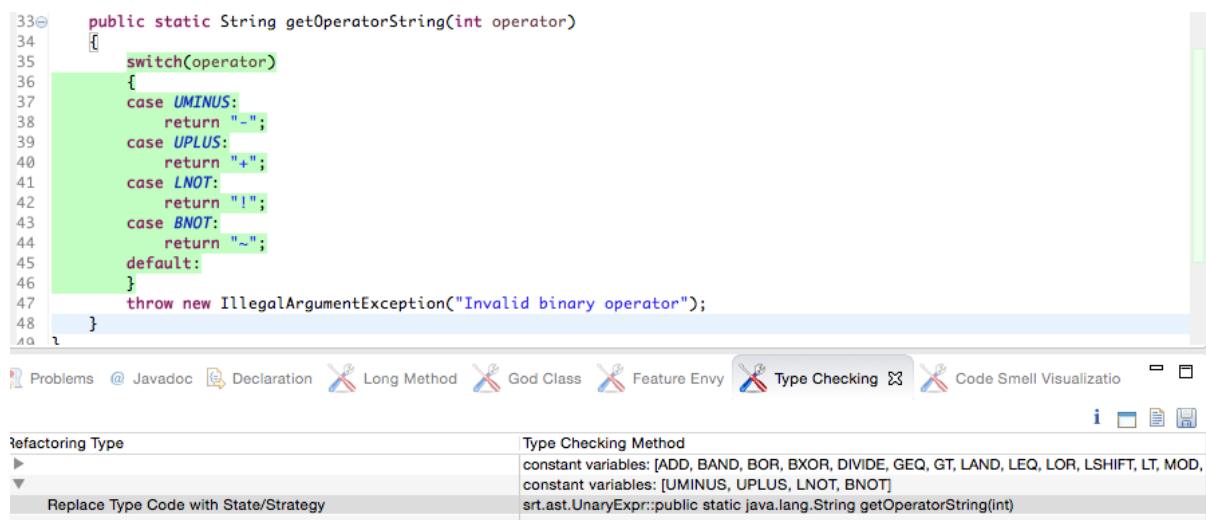


Figure 2.14: Eclipse JDeodorant plugin identifies state checking in source code

Refactorings are listed in the *JDeodorant* tool window interface and highlighted in the code when selected (see figure 2.14). We have found the highlighting functionality to be extremely useful, as it allows the user to easily determine which lines of the code are affected.

Once the developer chooses to apply one of the suggested refactorings, it applies the changes automatically. However, to keep the user informed about any changes that the plugin will make to the source code, a preview is available (see figure 2.15).

The preview gives users the opportunity not to follow through with the refactoring and/or find out what *JDeodorant* suggests before committing to the refactoring. We have found this preview to be a very valuable user interface element that greatly reduces anxiety when letting the tool refactor code automatically.

In this particular case, the plugin identified a translation between operators and strings as a refactoring opportunity and suggests the introduction of polymorphism to simplify the method. Notice, however, that even though the original method would be much easier to read after the refactoring, a large amount of complexity is introduced, as well as a `getOperatorObject()` method that resembles the old `getOperatorString()` method. One might therefore argue that this refactoring should not be performed.

Nevertheless, *JDeodorant* is one of the only available tools that is placed at the intersection of code smell identification and refactoring suggestions and shows a lot of promise. We believe that it would be even more useful if it were able to analyse code continuously to in-

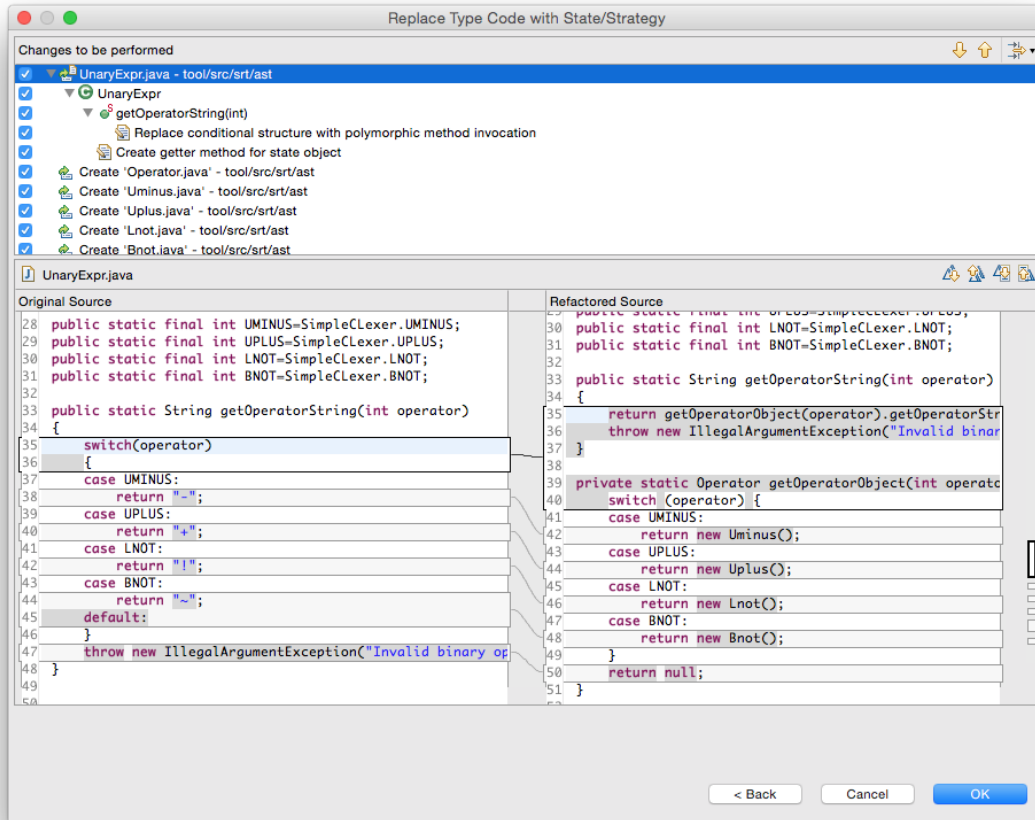


Figure 2.15: Eclipse JDeodorant plugin previewing refactoring steps

stigate a tight feedback loop between the developer and the tool. In addition, although the tool’s technical abilities are powerful, no explanations for why users should apply the suggested refactorings exist. While *JDeodorant* can help experienced developers apply refactorings more quickly, software engineers who wish to learn more about the problems in their code and why certain changes are performed will not find what they are looking for in this tool.

2.8 Code Structure Analysis

Various techniques to identify problems or design patterns are based on examining the structure of the code at hand. In the following sections, we will discuss some of these techniques.

2.8.1 Structure and Collaboration Templates to Identify Design Patterns

A range of different approaches to design pattern identification have been based on so-called structure and collaboration templates. The theory behind this approach is that most design patterns can be described in terms of the structure of the participating objects and their roles (collaboration).

Structure Pattern Pattern-specific constraints that a portion of a class diagram has to fulfil in order for it to belong to a pattern realisation [29].

Collaboration Pattern Description of the interactions between objects and classes that must exist for them to belong to a pattern realisation.

Together, the structure and collaboration templates can fully describe a design pattern. For example, table 2.1 shows a description of the well-known “Strategy Pattern”.

Structure	Collaboration
<ul style="list-style-type: none">• An abstract class must contain at least one abstract method declaration	<ul style="list-style-type: none">• An object must reference the abstract class• This object must call at least one of the declared abstract methods• Concrete implementations of the abstract class exist• The runtime type of the reference must be one of the concrete implementations

Table 2.1: Strategy pattern described using structure and collaboration templates

Bergenti and Poggi have developed a tool to identify design patterns using this technique. Their implementation uses Prolog¹⁵ rules to write the templates in logic form. Other developers are able to extend their rules [29].

By splitting the identification into structure and collaboration templates one is able to perform an efficient search through the available classes: only classes satisfying the structure template are used as a starting point for the matching of the collaboration templates.

Once a pattern is identified, improvements can be suggested. For instance, “Improving UML designs using automatic design pattern detection” [29] describes design critiques that can be suggested based on identified patterns: naming conventions or the different uses of access modifiers can be highlighted. An overview of this three-step process can be seen in figure 2.16

One of the most interesting contributions of this approach is that the naming of variables and methods does not have any effect on the identification of patterns. Rather, the search for patterns is guided by a more abstract description of the structure of the classes and indeed their interactions. Although *Vignelli* does not directly search for design patterns, we have based our approach to identifying potential problems in the source code on the same concept.

¹⁵<http://en.wikipedia.org/wiki/Prolog>, last accessed: 11 June 2015

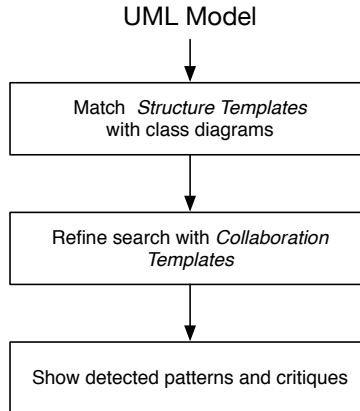


Figure 2.16: Pattern identification using structure and collaboration Templates

2.8.2 Finding Bug Patterns Using FindBugs

As discussed in section 2.7.2, *FindBugs* focuses on bug-finding techniques rather than the identification of higher-level design issues. Nevertheless, we discuss it here as the tool makes use of pragmatic techniques that have proved very valuable in efficiently reporting possible issues in code.

Traditionally, many bug-finding techniques have relied on formal methods and sophisticated program analysis to identify problems. Although these techniques make for a very thorough analysis, in practice, continuous and fast feedback are often more important during the development process. Instead of attempting to prove correctness of a full program, *FindBugs* therefore chooses to use a more pragmatic approach of identifying so-called *bug patterns*. In their paper “Finding bugs is easy,” Hovemeyer and Pugh describe bug patterns as code idioms that often result in errors [30].

Similarly to structure and collaboration templates (see section 2.8.1), *FindBugs* attempts to find bugs by examining the structure of the source code at hand and how objects interact. So-called “bug pattern detectors” describe a specific bug pattern in terms of the class structure and hierarchy, the control flow and the data flow. These bug pattern detectors then search for similar patterns in the source code that is being analysed by comparing it to the pattern descriptions. Although this static analysis technique is unsound and comparatively simple, the tool is popular in the developer community and has found countless bugs [30]. This shows that soundness is not required in all cases to make a useful tool to improve code quality.

2.8.3 Matrix-Guided Directed Search for Design Patterns

To optimise the search for design patterns in a large code base, Tsantalis and Chatzigeorgiou have developed a technique that combines the search of multiple individual design patterns into one [31]. The idea behind this approach is to focus on the common elements that some patterns exhibit.

In their paper “A Novel Approach to Automated Design Pattern Detection,” Tsantalis and Chatzigeorgiou describe how design patterns can also be described in terms of eight matrices and one vector, listing different properties about the structure of and collaboration between classes [31]. This formalisation of the structure and collaboration template approach makes it possible to easily find common elements in different design patterns.

Their idea is to build decision trees about design pattern realisations guided by the common

values in the matrices. If two patterns agree in parts of the matrices, they may belong to the same decision tree node. Search is performed from common root nodes and as the tree is traversed, options for design patterns are eliminated.

This approach of structuring the search for design patterns is intriguing, as multiple different design patterns can be searched for at the same time. Despite its potential performance advantages to other techniques, we currently do not use this technique in *Vignelli* as we cover no two similar patterns. We include it here, however, as we believe that this technique may be useful in future versions that support more problem identifications.

2.9 Metrics-Based Analysis

Direct identification of patterns by way of analysing structure and roles of programs is not the only way to identify code smells.

In their book *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*, Lanza and Marinescu describe metrics as a way to “characterise, evaluate, and improve existing code design” [32, p. 4]. They go on to explain that although good software metrics do not necessarily indicate a good design and bad metrics do not necessarily indicate bad code design, the use of metrics can be incredibly powerful when analysing code. One of the most popular applications of metrics is in visualisations of code design. This is because by combining different metrics, it is possible to not only visualise results and make them more accessible to the human eye, but also to use the numbers to identify potential flaws in the design.

Lanza and Marinescu go on to explain how only a few metrics can be combined in order to detect code smells in existing code bases. For example, the authors suggest that by using only four metrics (lines of code, cyclomatic complexity, number of variables, nested block depth [see section 2.9.1]), one is able to identify so-called “brain methods”, which are methods that contain so much logic that they are difficult to understand.

2.9.1 Commonly Used Metrics

Simple, commonly-used metrics include:

- LOC (number of lines of code in the method)
- NBD (nested block depth)
- PAR (number of parameters of a method)
- NOAV (number of accessed variables)

Cyclomatic Complexity

This metric measures the number of linearly independent paths through a program’s source code and is a measure of the complexity of a piece of code — the higher the complexity, the more execution paths exist through this method.

Cyclomatic complexity can be computed by considering the control flow graph of a method. Consider the code example in listing 2.14 which contains one `if`-statement and a `for`-loop.

The corresponding control flow graph can be seen in figure 2.17. Each node in the control flow graph represents one basic block, a straight-line piece of code without any jumps or jump

```

1 public void printMessage(String names[]) {
2     if (names.length == 0) {
3         System.out.println("No names!");
4     } else {
5         for (String name : names) {
6             System.out.println(name);
7         }
8     }
9 }

```

Listing 2.14: Simple Program containing Looping Control Flow

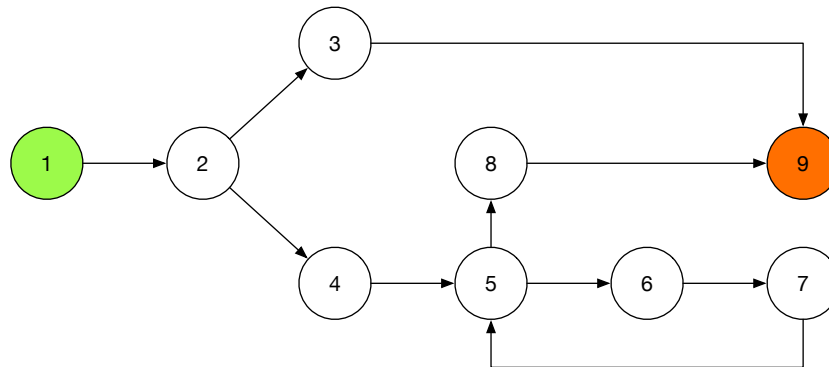


Figure 2.17: Control flow graph for code in listing 2.14

targets. Each edge represents a possible jump in the control flow of the program. In this example, the nodes are named after the line of code they correspond to.

Once the control flow graph of a program is computed, the cyclomatic complexity VG of a method can be calculated as

$$VG = E - N + 2$$

where E is the number of edges in the graph and N is the number of nodes in the graph. For example, the cyclomatic complexity for the program depicted in listing 2.14 is $10 - 9 + 2 = 3$.

2.9.2 Empirical Detection of Long Method Smell

One example of how metrics can be used to identify code smells is given by Bryton, Brito E Abreu, and Monteiro in their paper “Reducing subjectivity in code smells detection: Experimenting with the Long Method” [15]. In their approach, the researchers combine a number of metrics using *Binary Logistic Regression* in order to identify code suffering from the *Long Method* smell (see section 2.4.3).

Instead of attempting to spot patterns directly, the authors describe how they built a probabilistic classification model that can be used to predict to calculate the certainty with which a given method should be classified as a “long method”.

“Binary logistic regression (BLR) is used for estimating the probability of occurrence of an event [...] by fitting [previous] data to a logistic curve” [33]. The dependant variable only has two possible values, in this case *long method* or *not a long method*. The certainty with which a given method should be considered “long” can be expressed as the logistic function:

$$f(z) = \frac{1}{1 + e^{-z}} \quad z = \beta_0 + \beta_1 * x_1 + \beta_2 * x_2 + \dots + \beta_n * x_n$$

where x_i are the regressors and β_i are the regression coefficients. Many different metrics can be used as regressors. For the identification of “long method” smells, the following proved to be useful:

1. LOC (number of lines of code in the method)
2. NBD (nested block depth)
3. PAR (number of parameters of a method)
4. VG (Cyclomatic Complexity)

We notice that although Fowler and Beck describe the impact of comments in methods (see section 2.4.3), these do not feature in this list. As far as we know, no experiments have been conducted that take into account the number of comments or other associated metrics.

The probabilistic model for whether a method should be considered “long” can thus be expressed as

$$IsLongMethod = f(z) = \frac{1}{1 + e^{-z}} \quad z = \beta_0 + \beta_1 * LOC + \beta_2 * NBD + \beta_3 * VG + \beta_4 * PAR$$

Here, $f(z)$ describes the probability of the method under test being an instance of a *Long Method* taking into consideration its exposure to the risk factors z . The regression coefficients β_0, \dots, β_4 quantify the contribution of each of the metrics and need to be obtained from a training set of examples. In the paper, the authors describe their use of three expert users who classified a number of example methods from one project up front. Table 2.2 shows the coefficients that the authors were able to find based on their analysis of 193 different methods in one project.

Coefficient	Regressor	Value
β_0		-11.336
β_1	LOC	-0.057
β_2	NBD	4.701
β_3	VG	0.598
β_4	PAR	0.486

Table 2.2: Coefficients found based on analysis of one project

The resulting model was able to predict correctly 84% of the methods that are indeed *Long Methods*, and 99% of the methods which are not with a false positive rate of 6% and a false negative rate of 4% [15].

Although Bryton, Brito E Abreu, and Monteiro note that although this model was only based on the analysis of one project and therefore cannot be generalised, they were able to show that binary logistic regression can indeed be used to objectively detecting the “long method” smell.

To alleviate the problem of having to consult expert users, Pessoa, Abreu, Monteiro, *et al.* built on top of this approach in their paper “An Eclipse Plugin to Support Code Smells Detection.” The researchers built an *Eclipse* plugin that features a collaboration option where the knowledge of every user is collected and the model is adjusted on a central server and then pushed to clients on a repetitive basis [33]. Notably, this work was done under the assumption that a general BLR model exists that is not project-dependent.

2.9.3 Combinatorial Software Design Optimisation

Metrics are not only used to find potential issues in existing source code that can then be reported to users. Another application of metrics-based approaches are design optimisation techniques. These techniques use metrics and an understanding of the relationship between metrics and code design in order to optimise the design through well-known optimisation techniques.

For example, in their paper “Towards Automated Design Improvement Through Combinatorial Optimisation” [34], Cinnéide discuss the use of combinatorial optimisation techniques to improve the overall design of a software project automatically. Their technique is to reduce the problem of finding a class design with high cohesion, low coupling and good encapsulation, to an optimisation problem where the goal is the maximisation of a set of object-oriented design metrics.

The difficulty in this approach is to find adequate metrics and to evaluate them. Possible metrics discussed in the paper include, e.g., number of unused methods, number of featureless classes, or the number of abstract superclasses. These metrics then form a weighted sum which is then optimised using techniques such as simulated annealing to find close-to-optimal solutions.

This process takes a long time to run and is therefore not suitable for this project as we wish to give continuous feedback to the user. It is included here nevertheless to show the breadth of approaches that have been explored.

Chapter 3

Design and Implementation

Vignelli is an IntelliJ IDEA plugin that continuously observes the code a developer is currently working on, highlights code smells and suggests ways to refactor the code, in order to eradicate those smells.

3.1 User Interface Design

In order for the *Vignelli* project to be successful, it was necessary to develop an intuitive user interface that does not distract the developer from their current task in any major way but instead supports them in their programming.

In the *Vignelli* plugin this is primarily achieved by distinguishing between two *modes* the plugin can be in:

- Observation Mode
- Refactoring Mode

Switching between these two modes is done when the developer context-switches from a coding to a refactoring context.

3.1.1 Observation Mode

Observation mode is the standard mode that *Vignelli* is in during development. During this mode, the plugin observes what changes in the code are being made and attempts to spot bad smells. Being a continuous process, the developer does not have to explicitly run *Vignelli* on the code, but instead the plugin runs in the background, observing the code itself.

When a bad smell is detected in the code, it is highlighted using standard annotations in IntelliJ (see figure 3.1).

In this example, *Vignelli* has identified a potentially harmful method chain in the code and highlights it with a simple explanation. This is the minimal state in which the plugin can process in the background — developers are able to use this minimal UI of the plugin when they are already familiar with the different problems the tool can spot and no longer need more assistance or explanation.

Since *Vignelli* leverages the power of the IntelliJ inspection system, developers are able to have fine-grained control over which kinds of code smells the tool should highlight by enabling and disabling individual inspections in the IntelliJ settings. For example, a developer that is

```

void prepare() {
    Customer customer = new Customer();
    ZipCode zip = customer.getAddress().getZipCode();
    Label label = new Label();
    label.addLine(zip.toString());
}

```

This piece of code violates the Law of Demeter more... (⌘F1)

Figure 3.1: Vignelli inspection highlighting smelly code

familiar with train wrecks might wish to disable this inspection, yet keep all other warnings that the tool provides.

Less experienced developers that wish to learn more about the problems that *Vignelli* is able to highlight can enable an additional user interface element: an IntelliJ *tool window* at the bottom of the screen (see figure 3.2 for example).

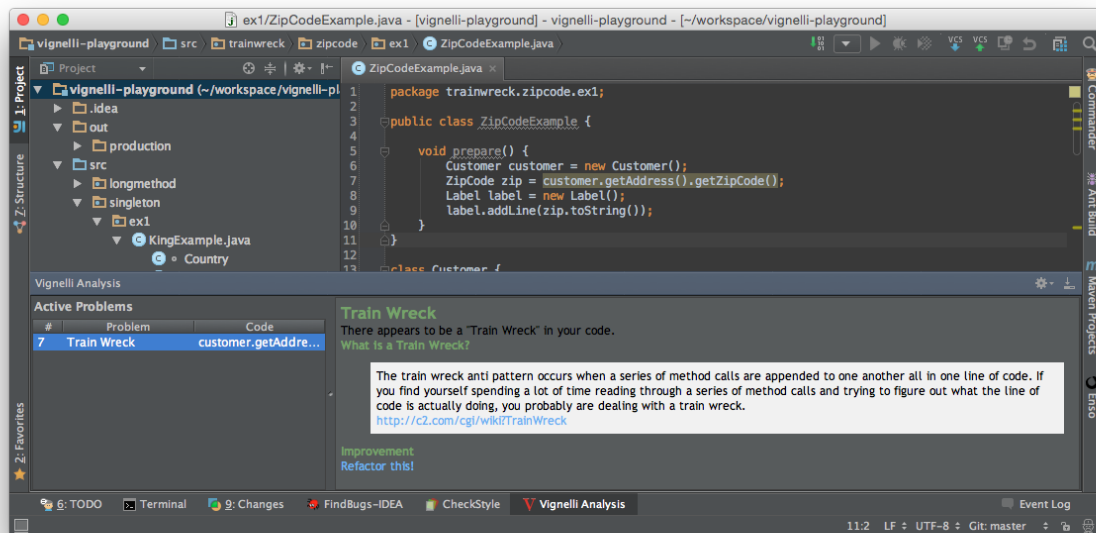


Figure 3.2: The *Vignelli* tool window explains problems in more detail

The tool window lists all problems in the file that is currently being edited by the user and provides detailed explanations of the potential problems on the right hand side. Since the tool window only lists problems in the current file, it is designed to allow developers to focus on the code at hand and prevent unnecessary context switches to other parts of the program — a problem that we found was prevalent in *JDeodorant* (see section 2.7.5).

3.1.2 Refactoring Mode

In the refactoring mode, the tool will assist developers in refactoring their code to eradicate a problem that was previously spotted by the tool. While the observation mode is targeted at developers of all levels, this mode is specifically designed to help developers who are unsure about how to move from a particular structure of the code to one that features a more future-proof design.

Users can start refactoring mode by clicking the “Refactor this” link below the problem explanation in the tool window and see if *Vignelli* is able to determine a way to refactor the

given code (see sections 4, 5, 2.4.3 for more details).

When clicked, the user interface in the tool window switches to focus on only one problem and on the steps required to resolve it.

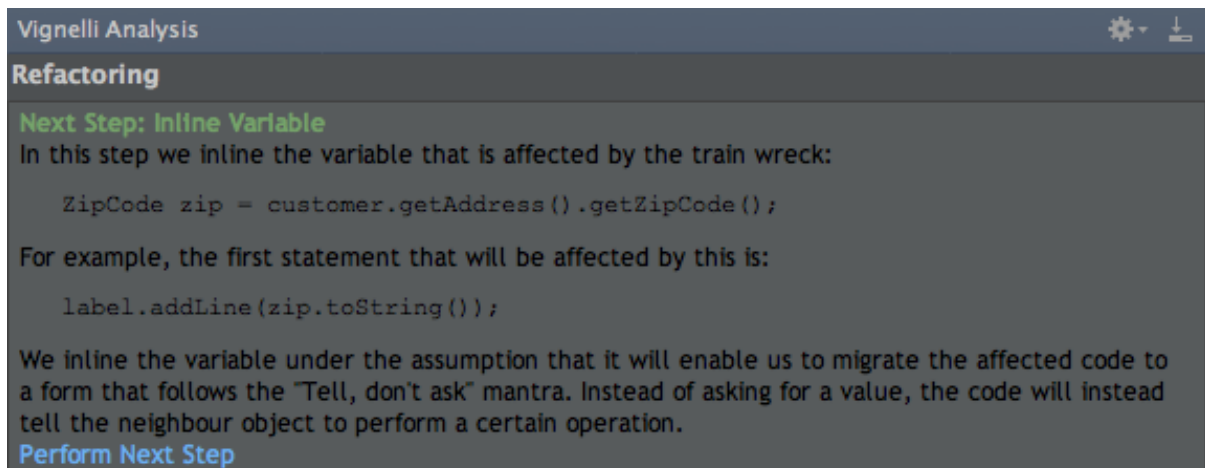


Figure 3.3: In refactoring mode, the tool window explains each step carefully

Figure 3.3 shows the explanation for one of the steps required in removing a train wreck. These explanations feature the actual code that is involved in the refactoring process and instruct the users of what needs to be done. Each step can then be performed by the developer by clicking the “Perform Next Step” link. In this case, clicking the link will bring up IntelliJ’s “Inline Variable” refactoring dialog which the user can then use to follow the instructions.

Note that the tool is designed for developer interaction and does not feature a “Quick Fix” mode in which problems can be solved in one step. Instead, the tool aims to teach its users how one would refactor the code in the hope that they will later know how to do so themselves.

Once the refactoring is complete, developers are presented with an overview of all the steps required and *Vignelli* switches back into observation mode.

3.1.3 Plugin Interaction and Developer Experience Level

Vignelli is designed to be used by developers of different levels of experience in software design and its user interface design aims to support this. Figure 3.4 illustrates how the plugin’s feature set is layered in that users can focus on simpler features as they gain experience.

As well as being alerted to potential problems, beginners may also require help during the refactoring process. As they gain experience with the particular problems, they are able to solely rely on the descriptions of the problems or even turn off the tool window altogether to only be alerted to problems via the IntelliJ inspection system.

3.2 System Architecture

Since the API for different IDEs such as IntelliJ and Eclipse differ significantly, *Vignelli* is specifically written as an IntelliJ IDEA plugin. As such, the architecture of *Vignelli* is heavily influenced by that of the IDE itself¹.

¹<https://confluence.jetbrains.com/display/IDEADEV/IntelliJ+IDEA+Architectural+Overview>; [last accessed: 1 June 2015]

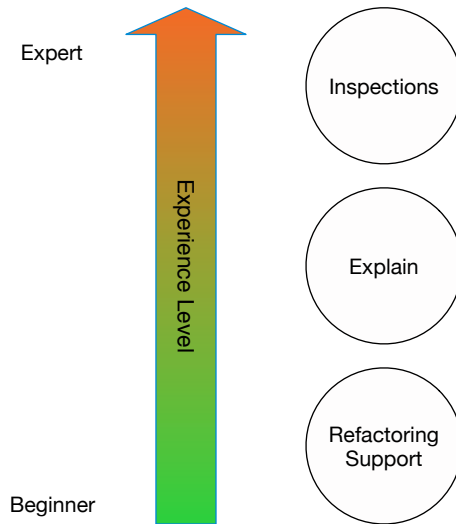


Figure 3.4: Different features appeal to users of different experience levels

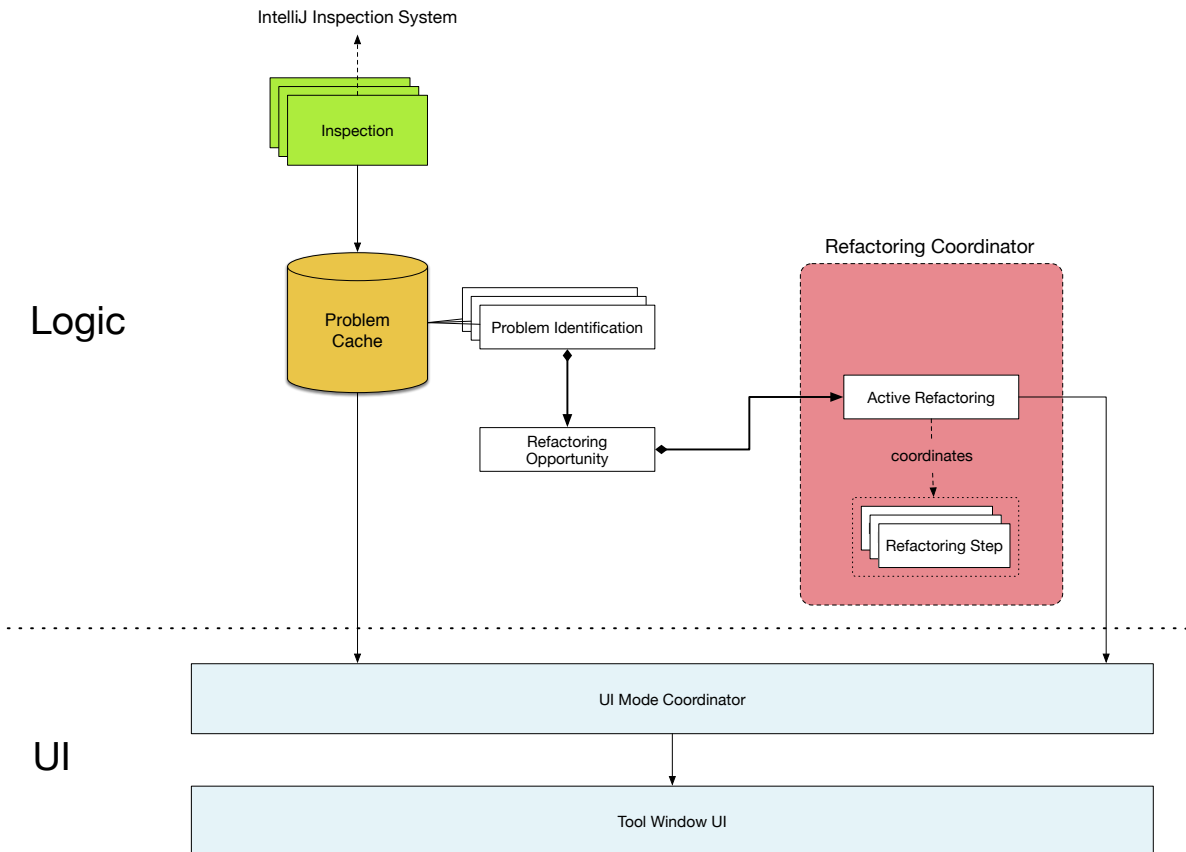


Figure 3.5: *Vignelli* architecture overview

As can be seen in figure 3.5, *Vignelli* harnesses the IntelliJ inspection system to power the identification of potential problems. Inspections allow IntelliJ plugins to analyse source code when it changes and provide problem descriptors to the IntelliJ instance that describe any problems that the plugin found. The IDE then uses these descriptors to highlight the relevant parts of the code in the editor and add a short description. For instance, one of the inspections that *Vignelli* defines identifies train wrecks and provides IntelliJ with problem descriptors that

highlight these train wrecks in the editor.

The problems that are identified in this process are also stored in the IntelliJ problem cache, which preserves the problem identifications for use in other parts of the plugin, namely the plugin-owned tool window user interface and any supported refactoring operations.

Since local IntelliJ inspections are run on every code change in the file, the problem cache is frequently updated with new problems. In addition to storing problem identifications, the cache also publishes changes to its data to any client that has subscribed to these changes (“publish-subscribe pattern”). This allows other parts of the system to receive up-to-date information on what problems currently exist in the code being edited.

One of the subscribers to this information is the user interface component. As updates about problems are received, the user interface updates the tool window to list only the most up-to-date problems with their respective descriptions.

Each problem identification that is stored in the problem cache may identify a so-called *refactoring opportunity*. These refactoring opportunities depend on the structure of the problem and are optional, i.e. *Vignelli* does not necessarily always provide a way of improving the code automatically. If a refactoring opportunity can be found, it is able to construct a description of a refactoring. Once constructed using all the required information, the refactoring opportunity can then start the refactoring process. This action will turn the refactoring into an *active* refactoring, coordinated by the refactoring coordinator.

Each refactoring is able to provide a rendered description of itself and the current refactoring step. These descriptions can be dynamically generated based on the code that is involved in each refactoring step.

As discussed in section 3.1.2, an active refactoring causes the user interface to switch and show information about the current refactoring, instead of a list of active problems. What is currently being shown is managed by the UI Mode Coordinator. If an active refactoring exists, the user interface subscribes to changes in the refactoring and re-renders it whenever the refactoring indicates a change (“observer pattern”).

In the following sections, we will discuss the individual components that make up the system in more detail.

3.2.1 IntelliJ Program Structure Interface (PSI)

The IntelliJ architecture revolves primarily around the program structure interface (PSI) that is used throughout the IDE’s own implementation and the APIs that are provided to plugin developers.

The PSI defines an abstract interface that allows developers to interact with the structure of the program that is being edited in the IntelliJ editor. Virtually all elements of the program structure are instances of `PsiElement`. For example, each file that contains code is an instance of a `PsiFile`. The PSI system is also used by IntelliJ to build abstract syntax trees (ASTs) of the programs that are being edited. These “tree representation[s] of the syntactic structure of [the] source code” [35] are used throughout the IntelliJ system to power all of its editing and refactoring functionality, i.e. the editor does not analyse or modify the textual representation of the code, but rather operates on this special structure which is generated and kept up-to-date automatically by the IDE itself.

In *Vignelli*, we therefore chose to use this built-in functionality to also support all of the plugin’s analysis and refactoring capabilities. Even though this renders the plugin more dependent on the IntelliJ platform and inhibits porting the application to other IDEs such as Eclipse in the future, using this AST instead of a custom tree greatly reduces the amount of work required. In

addition, it is easier to interact with IntelliJ’s built-in refactoring mechanisms of which *Vignelli* also makes use.

As mentioned above, IntelliJ automatically keeps track of changes to the code and modifies the tree structure accordingly. *Vignelli* makes use of this by incorporating various listeners for changes in the code. We also make extensive use of the “visitor pattern” to inspect the structure of the code.

Despite all of the advantages that the PSI system brings, we have also had to overcome various issues with the system. Most notably, since IntelliJ constantly updates the PSI tree as the code changes, parts of the tree frequently are replaced or removed. When a `PsiElement` is removed from the tree (e.g. because it was removed by the developer), it is still available to the plugin. However, the element is now marked as “invalid” which means that analysis or indeed refactoring operations involving this element can no longer be performed.

For example, figure 3.6 shows what happens when a parameter of a method is changed.

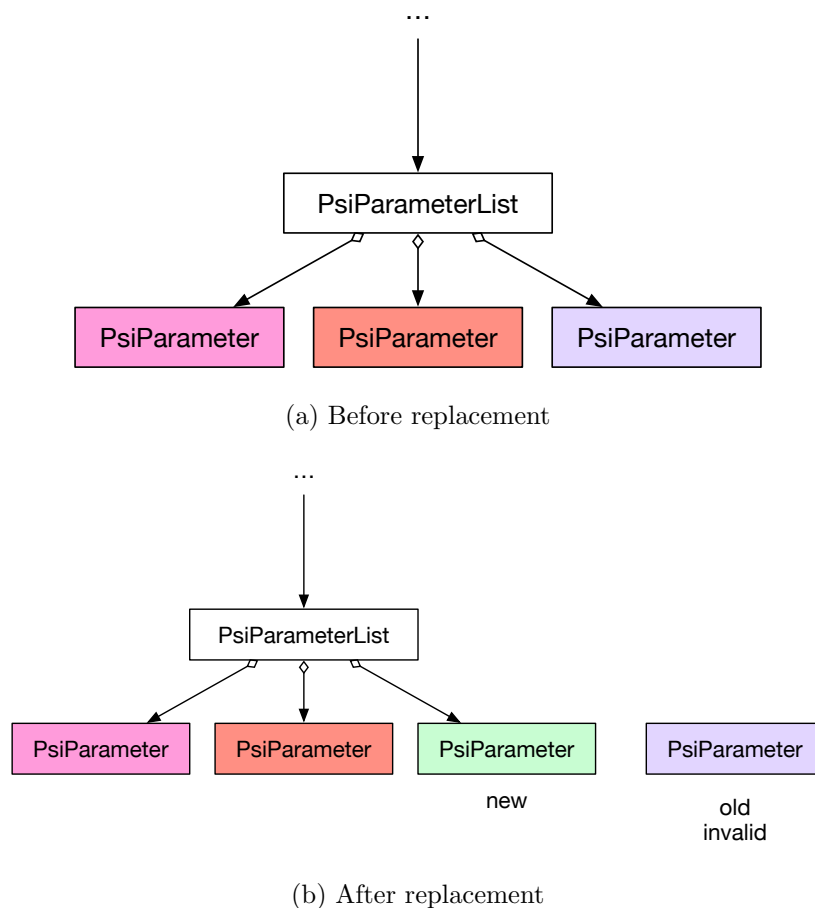


Figure 3.6: Parameter becomes invalid after changing it

Before replacement (3.6a), all parameters are associated with the method’s `PsiParameterList`. The third parameter (in purple) is part of the tree and it is therefore possible for a plugin to use it as part of the analysis and refactoring. For instance, a refactoring step that migrate the type of this parameter to a more general type is able to use this `PsiParameter` instance.

However, if the developer now changes the parameter manually (e.g. changing its name) the PSI tree is modified, the old `PsiParameter` instance is declared invalid, and a new `PsiParameter` instance replaces the old one in the tree. As we can see in 3.6b, the old instance still exists,

but is no longer valid. This complicates dealing with the PSI tree as the plugin can never be certain that elements are still able to be used and keeping track of all changes is difficult.

Nevertheless, this problem is in fact not caused by IntelliJ's PSI system; rather it is due to the nature of the continuity with which *Vignelli* should report problems in the code, update the user and provide helpful advice.

Vignelli's code and architecture is therefore complicated by having to handle invalidity of tree elements gracefully. In the following sections, we will cover particular cases where potential invalidity of elements was a problem and how it was dealt with.

3.2.2 Identification of Problems

Vignelli is designed to be extensible with various different code smell detectors. Each detector is responsible for identifying one type of problem. For example, *Vignelli* currently among others features a train wreck detector whose sole purpose is to spot train wrecks in the source code being edited.

As briefly discussed in section 3.2 problem detectors are built using IntelliJ's inspection system. In order to integrate with the rest of the plugin architecture and make it easy to add more inspections to *Vignelli*, we have written a set of classes and interfaces that make it easy to extend the system. Figure 3.7 shows an overview of how problems are identified in *Vignelli* and which classes/interfaces are involved in the process.

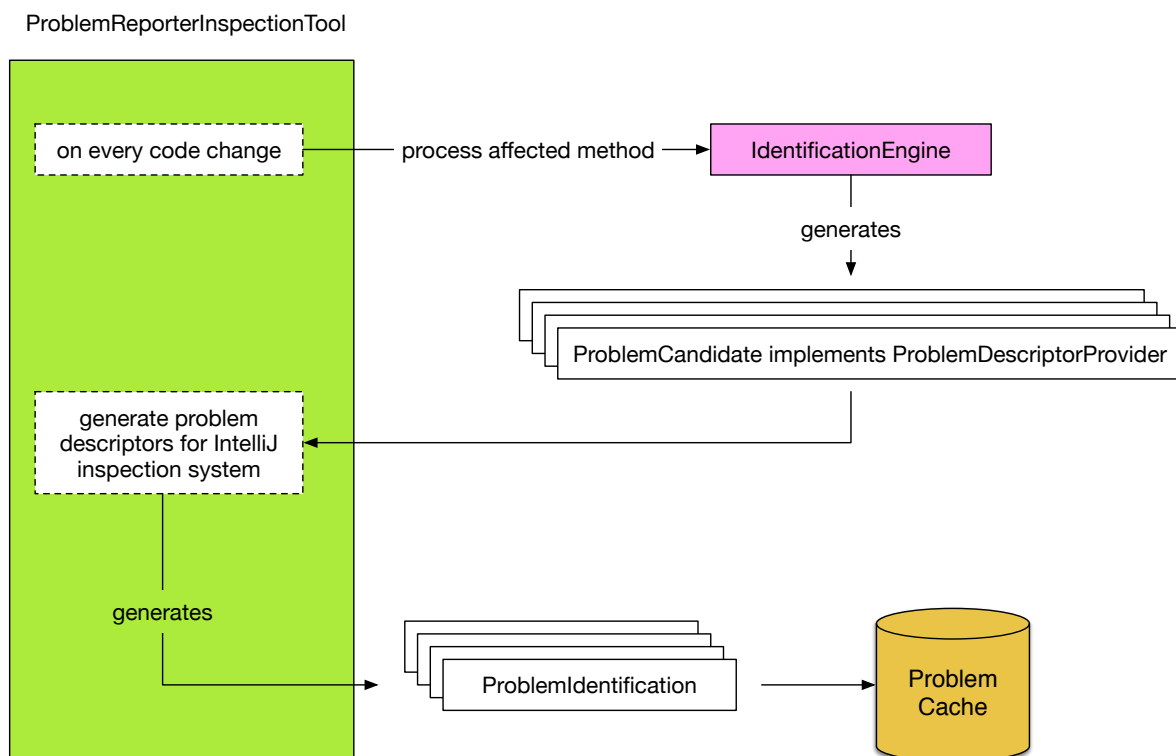


Figure 3.7: Problem identification overview

The starting point for every new code smell identification is the `ProblemReporterInspectionTool`. Implementations of this class coordinate the identification of problems and also ensure that problems are correctly forwarded to the problem cache, from where they will be broadcast to other parts of the plugin system.

Although code smell detection can technically be implemented here, *Vignelli* takes the approach of separating the identification logic from the inspection tool so that it can be reused in other places. More specifically, it is an `IdentificationEngine` (in pink) instance that is responsible for finding the actual problem candidates in the source code. Code smells are therefore detected in implementations of this interface. Since there may be multiple problems in one method — `IdentificationEngine` is called on a per-method basis whenever a method changes — a set of problem candidates is returned. Each problem candidate is able to generate `ProblemDescriptor` instances given an `InspectionManager`².

When all problem candidates have been found, it is up to the `ProblemReporterInspectionTool` to generate so-called `ProblemIdentifications`, which are descriptions of identified problems in the source code. These instances not only contain enough information for the plugin to later provide helpful refactoring suggestions, but are also renderable by the *Vignelli* UI system. In addition, `ProblemIdentifications` also are able to determine whether an improvement opportunity is available. These `ProblemIdentification` instances are subsequently stored in the problem cache.

The problem cache contains all active problem identifications. The data is organised according to which file the problems originated in, so that only those problems from the current file can be shown in the user interface. Whenever an inspection finds new problems, these are added to the cache and any old issues that no longer apply (and that were also detected by that same inspection tool) are removed.

Whenever changes are made to the problem cache these are then broadcast using IntelliJ’s built-in publish-subscribe system.

3.2.3 Launching Refactorings

The second main component of the *Vignelli* codebase corresponds to the plugin’s *refactoring mode* (see section 3.1.2). The plugin switches into this mode when a new refactoring becomes active.

Refactorings are launched by `ImprovementOpportunity` implementations. Figure 3.8 depicts how each problem identification can have multiple associated improvement opportunities. Often, there may be multiple different ways to solve an identified problem, using different refactoring techniques. It is up to the problem identification to decide which approach might be best-suited in that particular case.

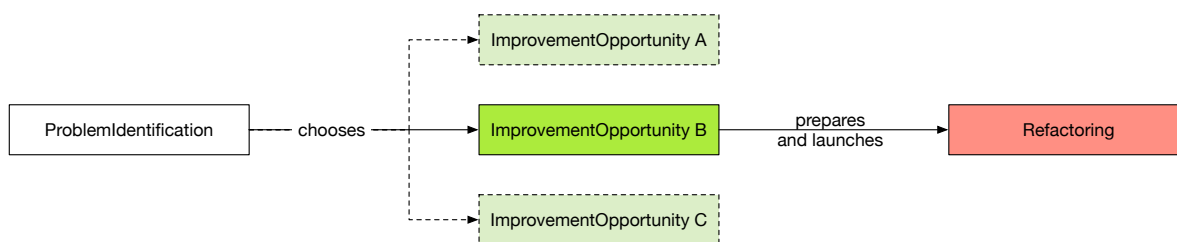


Figure 3.8: Problem identifications can launch different kinds of refactorings

It is the `ProblemIdentification`’s task to determine which `ImprovementOpportunity` will likely result in the best resulting code; to do this, the problem identification can analyse the PSI tree around the problem area. For example, *Vignelli*’s train wreck problem identification

²Inspection tools are given an `InspectionManager` instance by the IntelliJ system that allows the tool to generate `ProblemDescriptor` instances. Since the inspection manager instance is only available in the inspection tool instance we do not store it as part of the problem candidates.

decides between different improvement opportunities depending on whether a the result of a train wreck expression is assigned to a local variable or not (see section 4.2.2).

It is worth noting that it is entirely possible for the problem identification not to find any appropriate improvement opportunities, in which case none will be advertised to the developer via the tool window UI.

Once an improvement opportunity has been selected (`ImprovementOpportunity B` in figure 3.8), it acts as a factory and launcher for the corresponding refactoring. This means that this class is responsible for creating the refactoring with all the required parameters. Once this is done, it can then launch a refactoring.

Listing 3.1 shows an example of a trivial implementation of such an improvement opportunity. As we can see, it constructs the refactoring using the information required for launching the refactoring. `refactoring.begin()` causes the refactoring to be active. This is done by adding it to the given `RefactoringTracker`.

```
1 public class InternalGetterUseImprovementOpportunity implements ImprovementOpportunity {
2     @NotNull
3     private final PsiMethodCallExpression call;
4
5     public InternalGetterUseImprovementOpportunity(@NotNull PsiMethodCallExpression c) {
6         this.call = c;
7     }
8
9     @Override
10    public void beginRefactoring() {
11        Project proj = getterCall.getProject();
12        RefactoringTracker t = project.getComponent(RefactoringEngineComponent.class);
13        Refactoring refactoring = new InternalGetterUseRefactoringImpl(call, t, proj);
14        refactoring.begin();
15    }
16 }
```

Listing 3.1: Simple refactoring opportunity implementation

The `RefactoringTracker`'s main task is to coordinate active refactorings. Even though *Vignelli*'s current UI does not support multiple active refactorings at once, the `RefactoringTracker` does in fact support this. We had experimented with this functionality in previous versions of the tool; however, after some initial user testing, it was removed in favour of a simpler and clearer user interaction pattern (see section 7.3.1).

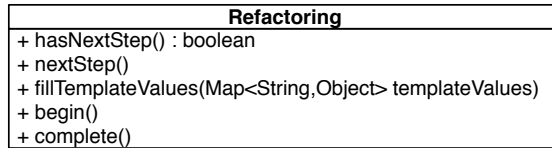
3.2.4 Refactorings and Refactoring Steps

Refactorings are implemented using primarily abstract classes/interfaces: `Refactoring` and `RefactoringStep`. Simplified versions of their UML diagrams can be seen in figure 3.9³.

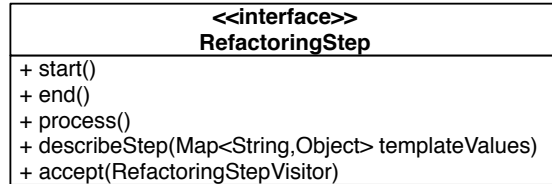
As mentioned in section 3.2.3 an `ImprovementOpportunity` creates a `Refactoring` instance and launches it; this is done by sending the `begin()` message (see figure 3.9a) to the `Refactoring` object. Likewise, refactorings can also be completed using the `complete()` message. This is typically done by a refactoring instance itself when it determines that the refactoring process has completed, but the message can also be issued externally, e.g. to cancel a refactoring from the UI.

Each *Vignelli* refactoring consists of one or more `RefactoringSteps`. For example, consider the case in which *Vignelli* has detected that the developer is calling out to a getter method inside

³The UML diagrams depicted here do not feature some of the constants that are declared in the implementations which are irrelevant to the discussion here.



(a) Refactoring UML diagram



(b) Refactoring step UML diagram

Figure 3.9: Simplified UML diagrams of the main refactoring components

its own class, when a reference to the corresponding instance variable would have sufficed. A **Refactoring** for this case could consist of only one **RefactoringStep** which replaces the getter call with the corresponding instance variable (see figure 3.10).

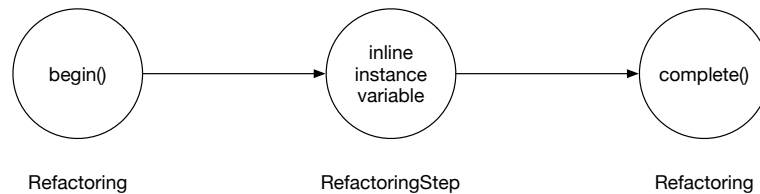


Figure 3.10: Internal use of getter refactoring flow example

A **RefactoringStep** describes one step in the refactoring process. However, rather than just acting as wrappers to IntelliJ’s own refactoring operations (such as the built-in refactoring to inline local variables), *Vignelli*’s refactoring steps not only provide a way to perform the task but also describe in broad terms what the goal of the refactoring is and are able to watch the PSI tree to look for changes that are relevant to that particular step.

3.2.5 Refactoring Step Goal Checkers

Figure 3.11 illustrates how *Vignelli* refactoring steps operate. Each step contains a so-called *goal checker*, whose task it is to search for a pattern in the PSI tree that would indicate that the operation required by this step has been completed. After a step is instantiated, its corresponding **Refactoring** therefore tells it to **start()** watching the PSI tree for changes.

Whenever the tree changes, the refactoring step’s goal checker (in red) inspects the current state of the tree with regards to some predefined success pattern and determines whether the goal state has been reached. If so, the corresponding refactoring is told about the change (using the delegation pattern), which can then perform the next required steps.

We developed the concept of *Vignelli*’s goal checkers for four reasons:

1. IntelliJ’s API that can be used by developers is limited regarding built-in refactorings. Although refactorings can be launched by plugins, there are very few ways in which plugins can pass options to control the outcome of a refactoring. For example, *Vignelli* is able

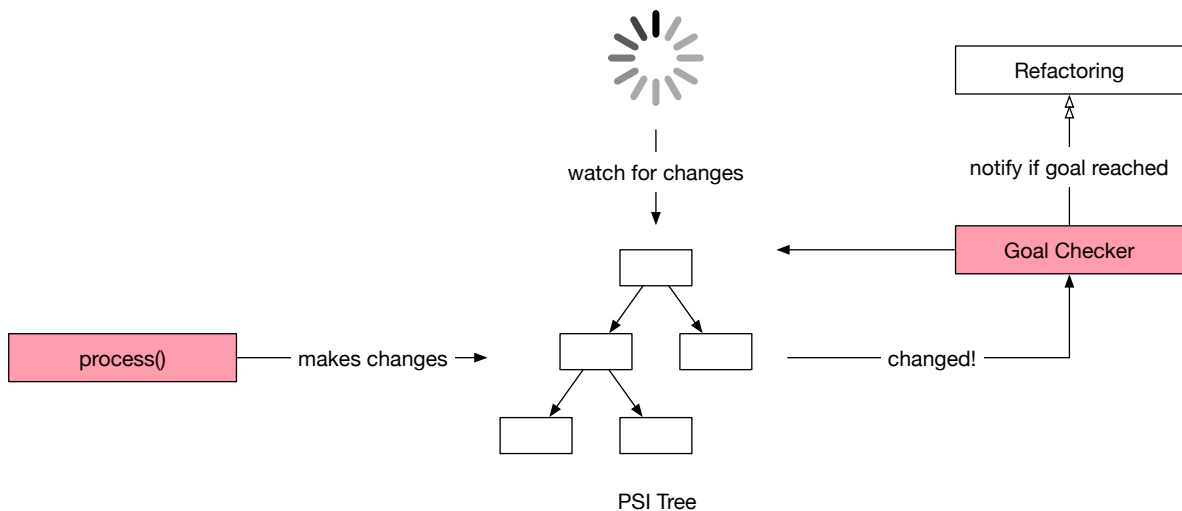


Figure 3.11: Refactoring step flow diagram

to launch a “Move Method” refactoring that moves a method from one class to another; however, it cannot control where the method is moved to. This means that even though *Vignelli* may launch a refactoring for a particular reason, the user may still disobey the plugin’s instructions and move the method elsewhere. *Vignelli* would be unable to pick up on these edge cases without the concept of goal checkers.

2. Likewise, IntelliJ also does not provide a universal way for plugins to be informed when refactorings are completed or indeed cancelled. It is therefore difficult for the tool to know when exactly a refactoring has been performed and the code is ready for the next refactoring step.
3. Using goal checkers allows users of the plugin to utilise IntelliJ keyboard combinations or context menus to trigger refactorings that are required by *Vignelli*. This adds to the learning process as developers will be able to use this workflow in other situations that *Vignelli* is not involved in, rather than clicking on a dedicated link inside the *Vignelli* tool window.
4. We have observed many beginners — in particularly those new to IntelliJ — refactor their code manually, without the help of IDE refactoring tools. Using the concept of goal checkers allows *Vignelli* to support this type of refactoring, which is beneficial to the plugin as a teaching tool for beginners.

However, it must be said that in practice, this support is limited and depends significantly on how the goal checker is implemented and in what way the developer modifies the code. This is because PSI tree modifications render old elements invalid (see section 3.2.1) and means that goal checkers can no longer use these elements in their checks. This in turn can lead to the abortion of the refactoring process as some of the elements required for checking the state of the program may no longer be valid.

In this project, we have experimented with two approaches to implement goal checkers:

1. Record which changes are made to the PSI tree and look for a predefined combination of certain actions
2. Inspect the PSI tree after every modification and attempt to recognise a predefined target code pattern

To illustrate how a goal checker might be implemented, consider *Vignelli*'s `InlineVariableRefactoringStep`, which describes a variable getting inlined. Before starting to look for changes in the tree, the goal checker records all of the statements in the code that will be affected by the inlining of the variable in question, i.e. all statements that refer to the variable to be inlined. Then, whenever the tree changes, the goal checker attempts to find any references to the variable in these statements. If none can be found, the variable was likely inlined.

Notice how the goal checker only approximates the check. In our example, one could trick the goal checker by adding an additional reference to the variable in another statement that previously did not feature a reference to the variable and then remove all references from those statements that were computed to be affected. Even though a reference to the variable and the variable itself will remain, the goal checker will notify the `Refactoring` that the goal has been reached. Despite being an approximation, user tests have shown that it is sufficient as users concentrate on the refactoring at hand and do not tend to deviate from the recommended process (see section 7.3.2).

3.2.6 Handling Refactoring Step Results

When a refactoring step's goal checker has determined that the step's particular goal state has been reached in the current PSI tree, its corresponding `Refactoring` is informed about this. In *Vignelli*, this is implemented using the *delegation pattern*, i.e. the `Refactoring` implements `RefactoringStepDelegate` (see listing 3.2).

```
1 public interface RefactoringStepDelegate {  
2     void didFinishRefactoringStep(RefactoringStep step, RefactoringStepResult result);  
3 }
```

Listing 3.2: The refactoring implements the `RefactoringStepDelegate` to be informed about step results

This means that each `RefactoringStep` is given its corresponding `Refactoring` as a delegate, which is informed about step completions via the `didFinishRefactoringStep` method. In order to keep track which step finished, it is passed as a parameter to the method along with a result object which can contain any information about what happened in the refactoring step. For example, each `RefactoringStepResult` defines whether the step has successfully completed or not. This allows the `Refactoring` to cancel the refactoring prematurely, e.g. if the relevant parts in the PSI tree that are needed by the goal checker have all become invalid.

For example, a simple way to implement the delegate method is shown in listing 3.3.

It is the `Refactoring`'s task to determine what is the next `RefactoringStep` to be instantiated and started, based on which steps have completed with which results. In order to avoid many instances of `instanceof` to determine which step has completed in the delegate method, the recommended approach in the *Vignelli* codebase to handle the result of a refactoring step and create the next step is to use the visitor pattern (see listing 3.4).

To ensure that `didFinishRefactoringStep` is only called once with the result and a new step is therefore only prepared and started once, *Vignelli*'s goal checker implementation ensures that the delegate is only notified once. This is to protect against the case when further changes to the PSI tree are made while the old step's goal checker is still running.

As we can see, this design allows the `Refactoring` implementation to determine which step should be taken next, not only depending on the result of the last step but also depending on which steps and which results were computed previously (provided the results are stored). This is a powerful mechanism and potentially allows *Vignelli* to adapt its refactoring suggestions if

```

1 public void didFinishRefactoringStep(RefactoringStep step, RefactoringStepResult res) {
2     if (!res.isSuccess()) {
3         complete();
4         return;
5     }
6
7     if (step instanceof InlineVariableRefactoringStep) {
8         // handle the result and create the next one
9     } else if (step instanceof ExtractMethodRefactoringStep) {
10        // handle the next result
11    }
12
13    // notify the UI to re-render the refactoring description
14    setChanged();
15    notifyObservers();
16 }

```

Listing 3.3: Simple way to coordinate different refactoring steps in the refactoring

```

1 RefactoringStepCompletionHandler handler = new RefactoringStepVisitorAdapter() {
2     @Override
3     void visitElement(ExtractMethodRefactoringStep extractMethodRefactoringStep) {
4         // use (ExtractMethodRefactoringStep.Result) result;
5         // handle result of the extract method refactoring step
6         // create next refactoring step
7     }
8     // ...
9 }
10 step.accept(handler);

```

Listing 3.4: Visitor pattern to handle results from different types of refactoring steps

the developer changes does not follow the recommended steps precisely.

Due to restrictions in the IntelliJ API, there are a number of limitations concerning some of the IntelliJ refactorings that affect the interaction between `RefactoringSteps` and `Refactorings`. Examples of these are outlined in section 3.3.2.

In summary, figure 3.12 depicts the general interplay between `Refactoring` and `RefactoringStep`. While it is the `Refactoring`'s task to determine which steps need to occur based on previous events and results, it is up to the `RefactoringStep`'s implementations to wait for the right conditions under which the step can be considered completed before they can notify the coordinating `Refactoring` object.

3.2.7 Undo Operation

IntelliJ's *Undo* operation lets users undo the last command they executed. These can be simple commands (e.g. the insertion of a character) or complex commands (e.g. an IntelliJ refactoring).

A *Vignelli Refactoring* consists of a number of refactoring steps. Each `RefactoringStep` is instantiated and started with a number of arguments, describing the PSI elements that should be targeted by the refactoring. For example, the `InlineVariableRefactoringStep` is passed the PSI element of the variable to inline when it is instantiated.

Unfortunately, in some cases IntelliJ's *undo* operation invalidates some of the elements that are required in the `RefactoringStep`'s goal checkers. Once invalid, *Vignelli* cancels the active refactoring — the tool does not attempt to migrate back to the previous step.

We hope to include more sophisticated logic about the state of PSI elements and the active refactoring in future versions.

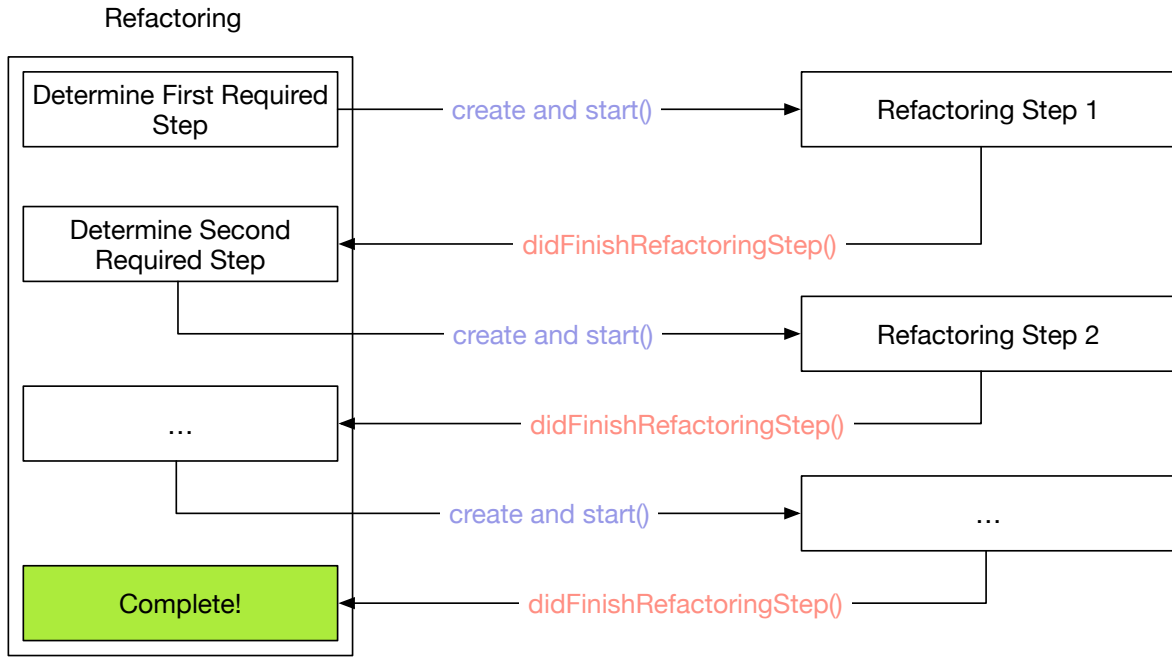


Figure 3.12: Refactoring and RefactoringStep interplay

3.2.8 User Interface Coordination

Vignelli's UI component uses the observer pattern to be notified of new problems that are found and any changes in an active refactoring (figure 3.13).

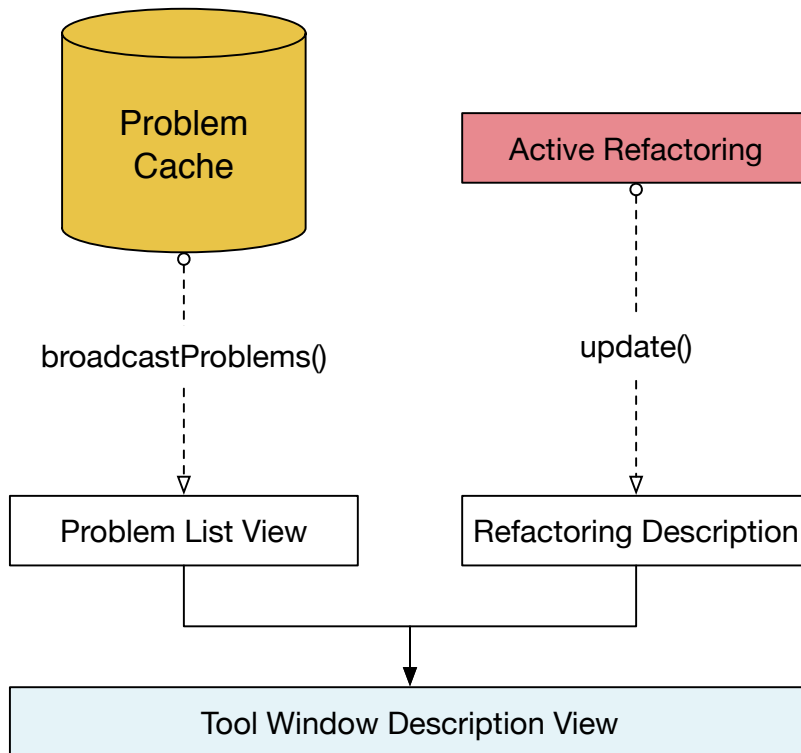


Figure 3.13: User interface coordination overview

When no refactoring is active — i.e. the problem list is shown in the tool window — the user interface updates the problem list every time changes are broadcast by the problem cache.

Once a refactoring is started, the user interface switches to the refactoring view. In particular, a new `RefactoringDescription` is created. This class acts as a renderable description of the refactoring and adds itself as an observer to the refactoring in question. Whenever the active refactoring is updated the description is re-rendered in the tool window.

Rendering of the description is a multi-step process that is shown in figure 3.14. The first step involves retrieving the description template for that particular refactoring from it. A *Vignelli* description template is an HTML template with placeholders that can be filled with concrete information. In the second step, the refactoring description tells the refactoring to fill in a map of template values. This map associates keys used in the description template with renderable objects. *Vignelli* then uses the simple *JMTE*⁴ template engine to render the template by replacing the placeholder keys with their corresponding values in the template.

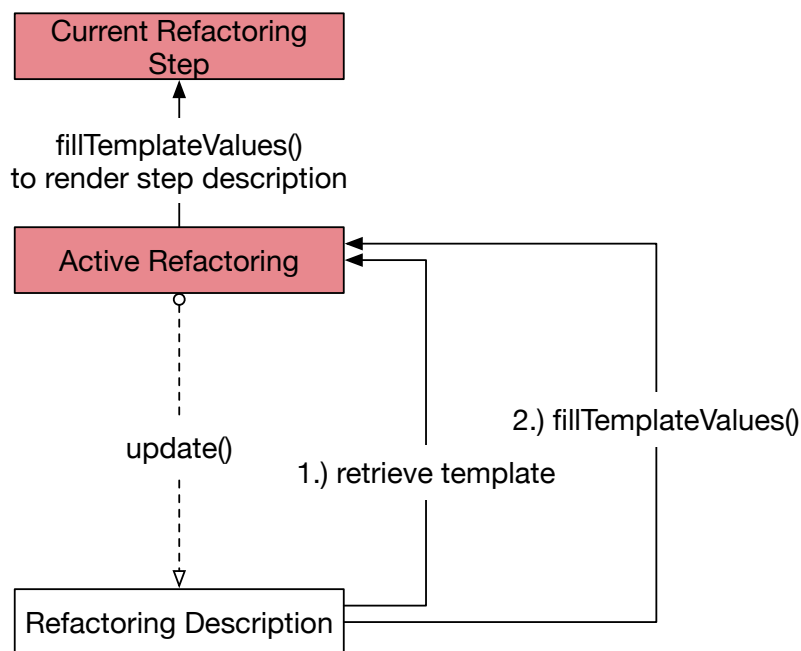


Figure 3.14: Rendering a refactoring

Although one may argue that template rendering process should be entirely contained in the refactoring implementation, there is a good reason for *Vignelli* to perform this task in the refactoring description; the tool is able to reuse the template values to display user interface elements corresponding to these values outside of the template. For example, one planned extension is to support a highlighting feature that allows users to hover their mouse over referenced code snippets in the tool window and have the corresponding code highlighted in the open editor.

Notice also that figure 3.14 shows the refactoring calling out to the currently active refactoring step to render itself. This allows the steps to define their own descriptions and fill parts of the refactoring description and supports the separation of concerns.

⁴<https://code.google.com/p/jmte/>; [last accessed: 1 June 2015])

3.3 IntelliJ IDEA Plugin Development

Even though IntelliJ provides a vast amount of APIs for developers, we have experienced a number of issues that were difficult to resolve given the design of some of the APIs and how we wanted to use them.

In the following sections, we will discuss two of the most prominent problems we faced interacting with the API that affected the overall code quality and functionality of the *Vignelli* plugin.

3.3.1 IntelliJ Refactoring Options

IntelliJ allows plugin developers to access APIs to launch IntelliJ refactorings to perform refactorings including, but not limited to, inline variables, extract methods and move methods.

These refactorings typically use user interface elements such as dialogs that let users choose options for the particular refactoring. For example, figure 3.15 shows the IntelliJ dialog that appears after launching the built-in “Move Instance Method” refactoring. It lets developers choose where the method should be moved to and what its visibility should be.

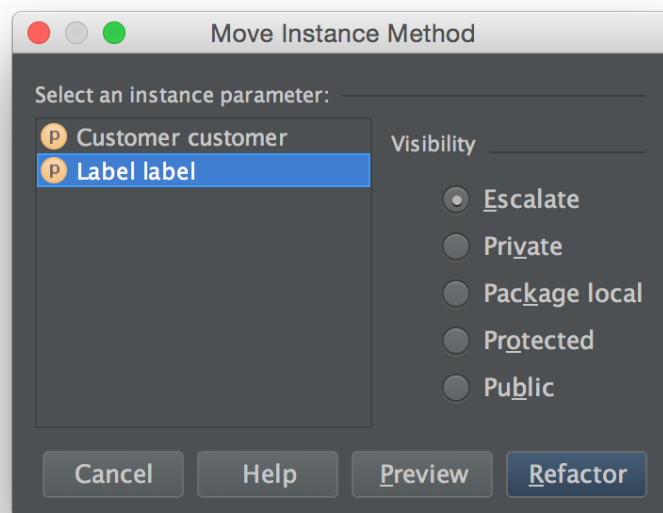


Figure 3.15: Move instance method refactoring dialog

As it turns out, it is not possible for plugins to launch this refactoring and restrict the options that are available to the developer. In particular, *Vignelli* is unable to prevent the user from choosing to move the instance method onto `Label label` when it should be moved to `Customer customer`. Users may also choose to change the resulting method’s visibility or even cancel the refactoring.

However, not only can plugins not restrict the dialogs but they can also not preset recommended options. For example, *Vignelli* is unable to preselect `Customer customer`. For the plugin this means that we have had to resort to writing clear instructions in the user interface on which options to select in the dialog boxes.

During user testing, this limitation became quickly apparent, however, to work around this

problem is outside the scope of this project (see section 7.3.2).

This limitation of IntelliJ is also one of the reasons for *Vignelli*'s concept of goal checkers which do not assume the usage of the built-in refactoring tools. This way, even if a refactoring has gone wrong, the developer still has a chance to resolve the problem and then move onto the next step.

3.3.2 Goal Checkers and PSI Tree Changes During Multi-Step IntelliJ Refactorings

Even though, in theory, *Vignelli*'s concept of goal checkers allows it to be independent of built-in refactoring procedures, this is not strictly the case in all cases in practice.

In particular, IntelliJ features a number of built-in refactorings that involve multiple steps during which plugins are unable to tell whether or not the IDE is currently performing one of these refactorings.

One particular example that also affects *Vignelli* code is that of the “Introduce Parameter for Expression” refactoring. This refactoring allows users to inject the selected expression as a parameter to the current method (see figure 3.16).

```

5 void prepare() {
6     Customer customer = new Customer();
7     Label label = new Label();
8     doAThing(customer, label);
9 }
10
11 private void doAThing(Customer customer, Label label) {
12     label.addLine(customer.getAddress().getZipCode().toString());
13 }
14 }

```

(a) Step 1: select expression to hoist into parameter

```

5 void prepare() {
6     Customer customer = new Customer();
7     Label label = new Label();
8     doAThing(customer, label);
9 }
10
11 private void doAThing(Customer customer, Label label, Address address) {
12     label.addLine(address.getZipCode().toString());
13 }
14 }

```

(b) Step 2: rename parameter

```

5 void prepare() {
6     Customer customer = new Customer();
7     Label label = new Label();
8     doAThing(label, customer.getAddress());
9 }
10
11 private void doAThing(Label label, Address theAddress) {
12     label.addLine(theAddress.getZipCode().toString());
13 }
14 }

```

(c) Step 3: finished introducing parameter

Figure 3.16: Introduce parameter refactoring

In *Vignelli* a goal checker exists to notify a refactoring that a parameter has been introduced.

The first step in the “Introduce Parameter” refactoring involves selecting the expression to hoist to a new parameter (figure 3.16a). When the IntelliJ refactoring is launched, it then introduces that selected expression as a parameter (figure 3.16b). However, notice that although the expression has already been introduced as a parameter, the refactoring is still ongoing with the editor waiting for the developer to rename the newly-introduced parameter.

As it turns out, even though the refactoring is still ongoing, the resulting PSI tree already features a new parameter. At the same time, there is no way for plugins to know whether the editor is in a state that requires the renaming of that parameter. This means that the goal checker identifies the state after step 2 as its goal state and notifies its delegate which will then launch the next refactoring step.

In step 3 of the refactoring process (figure 3.16c), the parameter is renamed. As discussed in section 3.2.1, renaming a parameter causes it to be replaced in the PSI tree and the old parameter to be `invalid`. The new refactoring will now be unable to use the parameter for further analysis as it is strictly no longer part of the PSI tree.

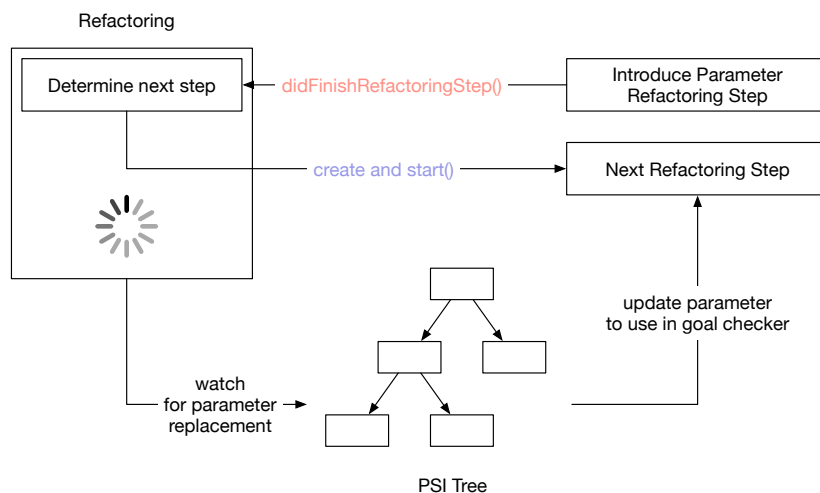


Figure 3.17: Watch for parameter replacements and update the parameter in subsequent refactoring steps

We have been unable to find an elegant solution to this problem in the *Vignelli* codebase and have resorted to a solution that creates a new PSI tree change listener that updates the next refactoring step’s reference to the parameter to any new version of the parameter in question that is introduced (see figure 3.17).

We hope that the IntelliJ development team will consider callback mechanisms for all refactorings that would allow us to find more elegant solutions.

3.4 Development Process

3.4.1 Continuous Integration

During the development of *Vignelli*, we have employed continuous integration techniques. This would be useful not only during this project in order to keep track of changes and test them early and often, but would also prepare the tool for external contributions once released to the public.

The first step in this process was to use version control to keep track of the changes. *Git* was chosen for this project because of its popularity in the open source community and the widespread tool support. The *Vignelli* *Git* repository is currently hosted on the Imperial-internal *Gitlab* platform⁵; however, we plan to move the software to *GitHub*⁶ after its release

⁵<http://gitlab.doc.ic.ac.uk/ss7311/vignelli>

⁶<https://github.com>

to open it up to a wider audience.

The second step in the process was to automate the software build process. IntelliJ plugins are typically developed inside a host instance of IntelliJ and can then be run inside a child process of the IDE using the built-in *IntelliJ Platform Plugin SDK*. Unfortunately, this way of developing is extremely dependent on each user using the exact same IntelliJ configuration. During the development of this project, we have even encountered issues with the IntelliJ-internal build breaking after IDE updates, which meant that the plugin could not be built until this was resolved.

In order to circumvent this issue, we developed a *Gradle*⁷ build script that allows developers to build the plugin from the command line using the typical *Gradle* commands⁸.

Since this approach of building IntelliJ plugins is officially supported by JetBrains⁹, writing the build script was initially a difficult task. However, using this script we were able to continuously build and test each change to *Vignelli* on a custom Jenkins CI server (screenshot in figure 3.18) — something of which, to the best of our knowledge, other plugins only rarely make use.

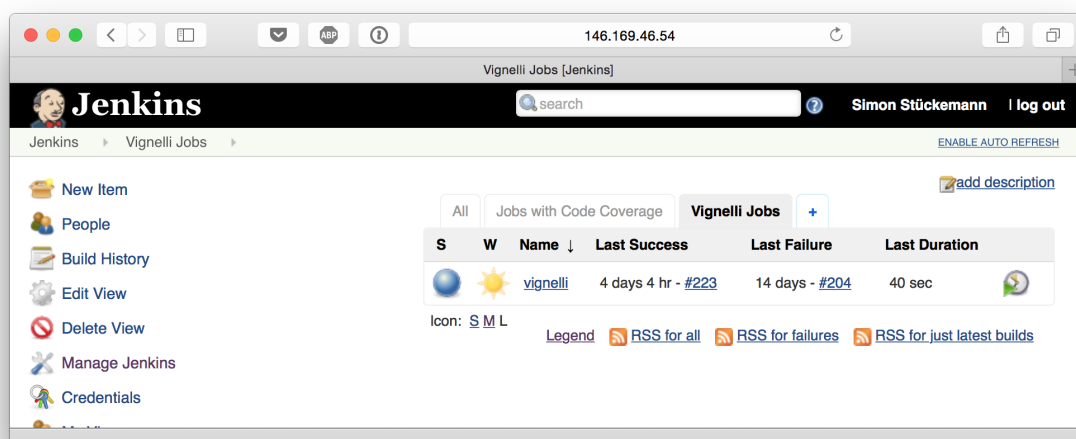


Figure 3.18: Jenkins job that builds and tests *Vignelli* automatically for every change

3.4.2 Testing

Tests have played a big role in our approach to use continuous integration throughout the project in order to ensure that the plugin works as expected and to minimise the number of bugs in software.

Testing Inspections and Identification Engines

We have used different approaches to test the identification of problems in *Vignelli* and the utility classes that we have written:

⁷<https://gradle.org>

⁸see <https://gitlab.doc.ic.ac.uk/ss7311/vignelli/blob/master/README.md> for more details on how to use *Gradle* to build the tool

⁹JetBrains are the IntelliJ developers

IntelliJ Inspection Tests

IntelliJ provides a testing framework that can be used by plugin developers to write comprehensive tests for the plugin’s functionality. According to the documentation, however, these are “model-level functional tests”. This means that: [36]

- “The tests run in a headless environment which uses real production implementations for the majority of components, except for a number of UI components.”
- “The tests usually test a feature as a whole, rather than individual functions that comprise its implementation.”
- “The tests do not test the Swing UI and work directly with the underlying model instead.”
- “Most of the tests take a source file or a set of source files as input data, execute a feature, and then compare the output with expected results (which can be specified as another set of source files, as special markup in the input file, or directly in the test code).”

The nature of these tests is perfectly suited for high-level tests that ensure correctness of the plugin’s behaviour inside the IntelliJ IDE. In particular, the test framework features easy ways to write tests for IntelliJ inspections in order to verify that the correct parts of the source code are highlighted.

An example of this is shown in listing 3.5. As we can see, Java code that should be highlighted by the IDE is wrapped in XML tags that describe the warning that should be generated.

```
1 // ...
2
3 ZipCode zip = <warning descr="This piece of code violates the Law of Demeter">customer .
4     getAddress() . getZipCode() </warning>;
5 // ...
```

Listing 3.5: IntelliJ test framework allows XML annotations inside Java code to test inspections

IntelliJ’s test system then runs the inspection on the code and ensures that the annotated warnings are indeed generated.

In *Vignelli*, we use this test API to write end-to-end tests for the identification of problems. We have written a number of example test cases that are all annotated according to what should be highlighted. IntelliJ’s test runner then verifies this desired outcome.

Converting Java Methods and Classes to PSI Trees

The test framework support for inspections checks each inspection in an end-to-end fashion, i.e. the entire identification process is run. However, during development we sometimes wish to test on a more local level which supports bug-finding.

For this reason, we have also used another technique to write test cases for more specific parts of the code. To do this, we read files that contains Java code snippets and parse them as either methods or entire classes in the test setup phase before converting these into PSI trees (see figure 3.19).

There are a number of advantages of using these tests in addition to IntelliJ’s test framework tests:

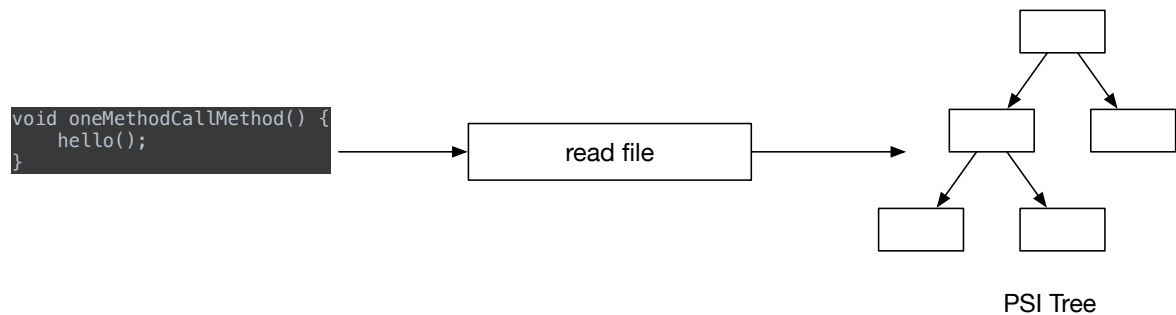


Figure 3.19: One method is converted into small PSI tree for testing

- We can test modules such as implementations of `IdentificationEngine` in isolation without the need to run the entire plugin.
- We can test individual methods rather than the entire process.
- Smaller PSI trees in tests result in easier debugging when tests fail.

Overall, this technique is a helpful addition to IntelliJ’s recommended ways of testing.

Mock Objects

Although we initially tried to use a test-driven development approach using techniques outlined in “Mock roles, not Objects” [20] using the popular *Mockito*¹⁰ test framework, we quickly discovered that it was too difficult to use mocking techniques with PSI trees. This is because PSI tree structures are extremely complex, and in order to have very simple interactions, many mocks may already be required.

We therefore quickly retreated to only using stubs to simplify some aspects of testing. For example, *Vignelli* contains a `ClassFinder` interface which defines methods to find classes in a given search scope. This is used to search for class definitions in the project scope. The real implementation searches the entire project for any open classes. However, some tests do not require this functionality. For this reason we use a fake version of this class finder. This simplifies the tests greatly.

Testing Refactoring and Refactoring Steps

Although we managed to write a number of very useful test cases for the problem identification modules of *Vignelli*, we have not been able to write useful tests for the `Refactoring` and `RefactoringStep` modules. This is due to a number of different factors:

- Each refactoring step’s manual invocation of the IntelliJ refactoring is not easily testable as these usually invoke some UI in the IntelliJ editor window. Interactions with these interfaces cannot easily be mocked in a headless test environment.
- Goal checkers depend largely on the structure of PSI trees. Changing PSI trees directly without IntelliJ refactoring operations is an extremely error-prone process. In addition, in order to try different kinds of interactions with the tree, one would have to change the tree in many different ways, making test code extremely verbose. However, we believe

¹⁰<http://mockito.org>, [last accessed: 1 June 2015]

that in the future, this might be solvable by using randomised testing techniques that modify the tree in random ways.

- **Refactoring** instances only create and start new **RefactoringStep** instances. Even though one could test whether the right refactoring is started, automating this process only makes sense in extremely complex decision processes which we have not encountered in this project. Instead, to determine whether the right steps are created and started at each step in the refactoring process, we prefer the use of manual testing at this time.

Therefore, although we are missing out on many of the benefits of automated testing, we believe that at the moment, the amount of effort required to write and maintain these automated tests far outweighs their benefit.

3.4.3 Documentation

Readme

To support other developers who wish to work on or just build *Vignelli*, we have written a README file that walks the reader through the setup process. It can be complicated to set up the plugin project for development in a local IntelliJ instance and a README file helps speed up this process. We have even found that using the file as a checklist also helps get the environment be set up faster than usual.

Javadoc

Since *Vignelli* is built for extension, we have made sure to document common interfaces and classes carefully using the *Javadoc*¹¹ syntax. Oracle's official tool can generate comprehensive documentation pages in HTML from these comments embedded between `/** ... */`. *Javadoc* allows developers to make useful annotations such as `@param` to describe each parameter to a method. These can then be browsed in the generated HTML pages from any web browser.

¹¹<http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html>

Chapter 4

Identification and Eradication of Train Wrecks

One of the problems that *Vignelli* is able to identify and eliminate through suggested refactoring steps is that of train wrecks, discussed in section 4. This chapter discusses the techniques that are used to realise this.

4.1 Identification

Following the *Vignelli* architecture outlined in section 3.2, the first step is to identify train wrecks in the source code. In this section, we discuss two approaches of developing a classification function and some of the edge cases in train wreck detection.

Both approaches are based on the ideas behind structure and collaboration templates (see section 2.8.1). Pre-defined templates of the structure of train wrecks and the interactions between the objects that are involved are used to pattern match train wrecks in the source code.

4.1.1 Simple Approximation: Multiple Call Chains

Since train wrecks are first and foremost chains of method calls, a first approximation of a train wreck classification function is one that counts the number of chained method calls.

Consider the method call chain depicted in listing 4.1.

```
1 ZipCode zip = customer.getAddress().getZipCode();
```

Listing 4.1: Simple train wreck example

As we can see, `customer` is asked for its `Address`, which in turn is asked for its `ZipCode`. This method chain involves two method calls (`getAddress()` and `getZipCode()`) and one variable that acts as a method call qualifier (`customer`).

In its simplest form, a train wreck classification condition could therefore look as follows:

$$\text{isTrainWreck}(\text{expression}) = \text{number of method calls in expression} \geq 2 \quad (4.1)$$

However, there are a number of problems with this simple condition. One of these is exposed by the code in listing 4.2, which also features an expression that consists of two method calls.

Notice, however, that no method call qualifier in form of a variable (like `customer` in the previous example) exists. The simple train wreck classification condition 4.1 would classify this method call chain as a train wreck.

```

1 class ZipCodeExample {
2     Customer customer = new Customer();
3
4     void prepare() {
5         Address address = getCustomer().getAddress();
6     }
7
8     void getCustomer() {
9         return customer;
10    }
11 }

```

Listing 4.2: Helper methods should not count towards train wrecks

In this case, though, `getCustomer` is a helper method rather than a neighbour's method, which means that `getAddress()` actually asks `ZipCodeExample`'s neighbour for data — a method call chain that should not be highlighted as a train wreck.

For this reason, we can adapt the train wreck classification condition to take into account any variable qualifiers such as `customer`.

$$\text{isTrainWreck}(\text{expression}) = \# \text{ method calls and variable qualifiers in expression} \geq 3 \quad (4.2)$$

Since the number of method calls and variable qualifiers in the example expression in listing 4.1 is 3 overall, the condition will now still classify this example as a train wreck, while the use of a helper method in listing 4.2 is no longer considered a train wreck.

These two examples show that even though train wrecks are often described in terms of method call chains only, it is not sufficient to only use method calls to identify train wrecks.

4.1.2 Train Wreck Classification in the Context of Fluent Interfaces

Having defined a simple train wreck classification condition in section 4.1.1, now consider the `Request` class definition from section 2.4.1 given here again for convenience in listing 4.3.

The `Request.Builder` is used to create a new `Request` instance. We can now attempt to classify whether or not the method chain contained in this code snippet is a train wreck using train wreck classification condition 4.2. This results in the following classification:

Contains Variable Qualifier	Yes
Number of Method Calls	5
Classification based on classifier 4.2	Train Wreck

Evidently, classification condition 4.2 determines that the expression of the form `builder.withUrl("http://example.com").withContent("").withAttempts(10).build()` should be considered a train wreck.

However, as discussed in section 2.4.1, this code actually does obey the Law of Demeter, i.e. the `BuilderExample` only communicates with its neighbouring `Request.Builder` object in the object structure graph (see figure 2.5 in section 2.4.1 for an illustration). This shows that although classification condition 4.2 is able to classify actual train wrecks correctly, it will also

```

1 public class BuilderExample {
2     public void execute() {
3         Request.Builder builder = new Request.Builder();
4         builder.withUrl("http://example.com").withContent("").withAttempts(10).build();
5     }
6 }
7
8 class Request {
9     private final String url;
10    private final String content;
11    private int attempts;
12
13    Request(String url, String content, int attempts) {
14        this.url = url;
15        this.content = content;
16        this.attempts = attempts;
17    }
18
19    public int getAttempts() {
20        return attempts;
21    }
22
23    public void send() {
24        attempts--;
25    }
26
27    public static class Builder {
28        private String url;
29        private String content;
30        private int attempts;
31
32        public Builder withUrl(String url) {
33            this.url = url;
34            return this;
35        }
36
37        public Builder withContent(String content) {
38            this.content = content;
39            return this;
40        }
41
42        public Builder withAttempts(int attempts) {
43            this.attempts = attempts;
44            return this;
45        }
46
47        public Request build() {
48            return new Request(url, content, attempts);
49        }
50    }
51 }

```

Listing 4.3: Request provides a builder with a fluent interface

report false positives in the context of fluent interfaces through method chains, as is the case when using the builder pattern.

In order to address this problem adequately, we would therefore have to take into account the object navigation structure, rather than the method calls contained in the method call chain.

4.1.3 Computing the Object Navigation Structure

The object navigation structure inherently depends on the runtime configuration of the program under consideration. This is because instead of considering the relationships between classes only, the object navigation structure depends on the instances of those classes at runtime. As

it turns out, it is not always possible to statically determine this structure.

To illustrate this, consider the code in listing 4.4. This code generates two nodes, `node1` and `node2`. The program then randomly assigns either `node1` itself or `node2` to `node1`'s `next` field. The code then queries the `data` field of `node1`'s `next` node via the method call chain `node1.getNext().getData()`.

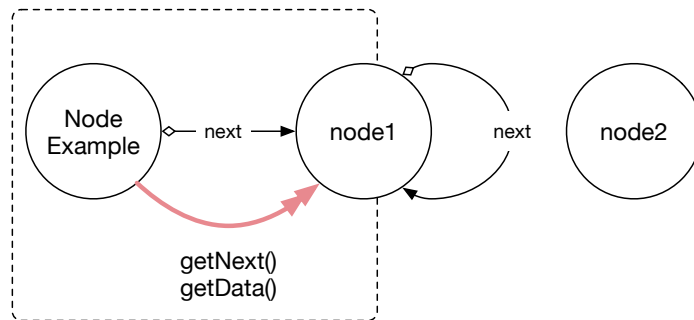
```
1 class NodeExample {
2     public static void main(String[] args) {
3         Random rand = new Random();
4         Node node1 = new Node();
5         Node node2 = new Node();
6
7         // generates 0 or 1 randomly
8         int choice = rand.nextInt(1);
9
10        if (choice == 0) {
11            node1.next = node1;
12        } else {
13            node1.next = node2;
14        }
15
16        // retrieve data from the second node, whichever that is.
17        node1.getNext().getData();
18    }
19 }
20
21 class Node {
22     Node next;
23     Object data;
24
25     Node getNext() {
26         return next;
27     }
28
29     Object getData() {
30         return data;
31     }
32 }
```

Listing 4.4: Random object assignment has effect on object navigation structure

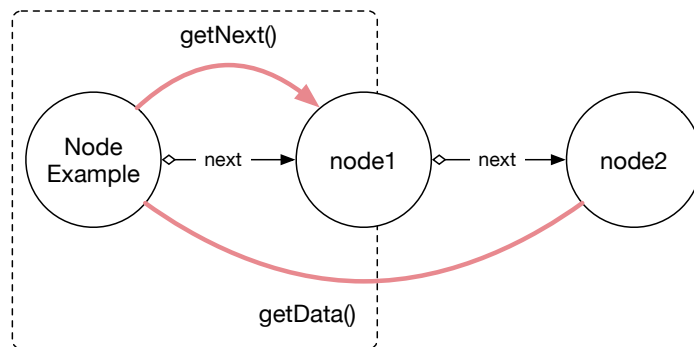
Notice, however, that the result of `getNext()` depends on the random number which is generated at runtime. This fact leads to two distinct object navigation structures for the method call chain that are shown in figure 4.1. Should `node1`'s `next` field contain a reference to `node1` itself (see figure 4.1a), the method call chain should not be classified as a train wreck. However, it is also possible for `node1`'s `next` field to contain a reference to `node2`. In this case, the object navigation structure (see figure 4.1b) suggests that the code should be classified as a train wreck.

This example illustrates that we are unable to compute the object navigation structure for method call chains with 100% accuracy through static analysis. Since *Vignelli* runs as an IntelliJ plugin and is required to analyse code statically¹, the tool will therefore also not achieve 100% accuracy in its classification results.

¹This is due to *Vignelli*'s reliance on the IntelliJ PSI tree and the inspection system.



(a) Object navigation structure suggests no train wreck



(b) Object navigation structure suggests train wreck

Figure 4.1: Object navigation structure depends on random outcome

4.1.4 Static Approximation for Object Navigation Structure

Since we are unable to accurately determine the object navigation structure of a method call chain in *Vignelli*, the plugin instead computes an approximation of the structure.

The approximation is based on the static return types of the method calls that make up the method call chain. To illustrate this, consider figure 4.2. In this illustration, we have highlighted the individual method calls that make up the method call chain, according to the static return type the corresponding method.

```
builder . withUrl("http://example.com") . withContent("") . withAttempts(10) . build()
```

Request.Builder
 Request

Figure 4.2: Request builder method calls' static return types

As we can see, this method call chain contains method calls of two different types: `Request.Builder` (every call and variable qualifier except `build()` and `Request` (for `build()`). Notice also that as the method calls are executed (first `withUrl("http://example.com")`, followed by `withContent("")`, etc.), the static type only changes **once**: when `build()` is called.

As an approximation to constructing the object navigation structure, *Vignelli* therefore makes the assumption that no change in static type from one method call to the next is equivalent to no change in the object to which the message is sent. In the running example of the `Request.Builder`, *Vignelli* therefore assumes that the builder's `withUrl`, `withContent`, and `withAttempts` methods return `this` where `this` is the instance that is executing said method.

Based on this observation we define the *type difference of a method call chain* in the following

way:

Definition 1. *The type difference of a method call chain is defined as the number of times the static return types of adjacent method calls in the chain differs.*

According to this definition, we can calculate the *type difference* for a number of example method call chains (including the example in figure 4.2). A number of example results can be found in table 4.1.

Method Call Chain	Static Types	Type Difference	Train Wreck
<code>builder.withUrl("http://example.com").withContent("").withAttempts(10).build()</code>	A:A:A:A:B	1	no
<code>builder.withUrl("http://example.com").withContent("").withAttempts(10).build().getAttempts()</code>	A:A:A:A:B:C	2	yes
<code>customer.getAddress().getZipCode()</code>	A:B:C	2	yes
<code>node1.getNext().getData()</code>	A:A:B	1	sometimes
<code>customer</code>	A	0	no

Table 4.1: Type difference for example method call chains

Based on these results we can define an alternative train wreck classification condition that uses the type difference approximation:

$$\text{isTrainWreck}(\text{expression}) = \text{type difference of expression} \geq 2 \quad (4.3)$$

As we can see, this approximation will *always* accurately classify three of the four examples in table 4.1. The `node1.getNext().getData()` will only *sometimes* be classified correctly.

4.1.5 The void Type

As explained in section 2.4.1, one way to avoid train wrecks when writing code is to follow the “Tell, don’t ask” directive [13] and instruct neighbours to perform a certain task, rather than ask for data and act on it.

With this in mind, consider the example given in listing 4.5.

```
1 builder.withUrl("http://example.com").withContent("").withAttempts(10).build().send();
```

Listing 4.5: void method call on builder result

As before, this code constructs a new `Request` instance using the builder pattern. However, the resulting `Request` is subsequently sent the `send()` message. The corresponding `send` method has static return type `void`.

According to classification condition 4.3, this method call chain would be classified as a train wreck. However, we can make the following observations about `void` methods in the context of method call chains in general:

- Since methods of type `void` do not return data, calls to such methods can only occur at the end of a method call chain.

- Since methods of type `void` do not return data, clients that call this method do so in order to instruct the callee to perform an operation. In other words, callers *tell* callees to perform an action, as opposed to *asking* for data.

Based on these observations, we can interpret the code in question as an example of the client *telling* a newly-constructed object to perform an action — something that should not be considered a train wreck.

This interpretation of the code is reinforced by the fact that alternative ways to design this code do not make sense in the context of the builder pattern; for instance, telling the `Request.Builder` instance to record a `send` attempt cannot be justified semantically in this context.

Therefore, when classifying potential train wrecks, we can take into account whether or not the final call in a method call chain is of type `void`. To do this in *Vignelli*, we adapt the definition for the `type difference`:

Definition 2. *The type difference of a method call chain is defined as the number of times the static return types of adjacent method calls in the chain differs. Type differences where one of the types is `void` do not count towards the type difference.*

According to the new definition 2, the type difference of the method chain in listing 4.5 is still 1, meaning that this method call chain should not be interpreted as a train wreck according to classification condition 4.3.

Now, consider the following example in listing 4.6.

```

1 // Method chain in question
2 customer.getAddress().remove();
3
4 // Modified class definition of Address contains method to remove itself
5 class Address {
6     // ...
7     void remove() { ... }
8 }

```

Listing 4.6: `void` method call on non-builder method call chain

According to the new definition 2 the type difference of this method call chain is 1. Thus, according to train wreck classification condition 4.3, this method chain should not be considered a train wreck.

However, now consider the associated object navigation structure in figure 4.3.

Clearly, `OrderDisplay` communicates with `Address`, which is outside of its immediate neighbourhood. According to this structure, the example should be classified as a train wreck.

We therefore further modify our train wreck classification condition to take this case into account:

$$\begin{aligned}
 \text{isTrainWreck}(\text{expression}) &= \\
 &= \\
 &= \text{type difference of expression} > 1 & (4.4) \\
 &\vee \\
 &= \text{type difference of expression} = 1 \wedge \text{length of expression} = 3
 \end{aligned}$$

With the new classification condition 4.4, example 4.6 will now be classified as a train wreck due to the special clause taking the length of the expression into account.

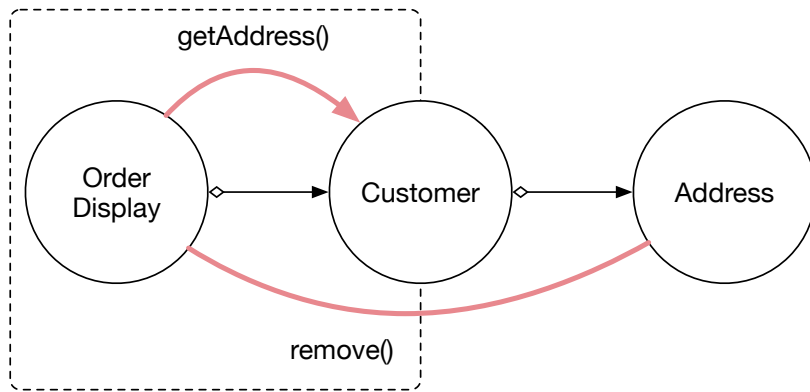


Figure 4.3: Object navigation structure for short method chain with void final call

4.1.6 Support for Static Calls

Train wreck classification condition 4.4 assumes that all method calls that make up the method chain to classify are calls to instance methods of a class — calls to static methods have not been considered so far.

Listing 4.7 shows code that uses the singleton pattern to retrieve the one and only King instance and then retrieve a set of peasants from said king.

```

1 class StaticCallsExample {
2     void reward() {
3         Bank bank = ...
4         bank.giveCoins(King.getInstance().getPeasants());
5     }
6 }
7
8 class King {
9     static King getInstance() { ... }
10
11     Set<Peasant> getPeasants() { ... }
12 }
13
14 class Peasant { ... }
15
16 class Bank {
17     void giveCoins(Set<Peasants>) { ... }
18 }

```

Listing 4.7: Singleton retrieval via static call leads to method chain

The method call chain `King.getInstance().getPeasants()` has type difference 2 and can therefore be classified as a train wreck. Indeed, one can argue that this method call chain is a train wreck and should be highlighted to the developer.

However, in *Vignelli* we have decided to exclude train wrecks due to static calls for the following reason:

In our previous arguments for classifying method call chains as train wrecks, we have referred to the object navigation structure in terms of the client object’s neighbourhood. In the context of static calls, this concept of neighbourhood can no longer be clearly defined as no aggregation relationship exists between the classes. Instead, the classes are related purely based on method calls — the relationship does not fit into a purely object-oriented model. Indeed, there has been much debate in the OOP community about the merits of `static` classes and methods [37]–[39]. In this argument, we personally side with the view that using the `static` too liberally leads to

code that no longer benefits from many of the advantages of object orientation.

One way to remove the train wreck from the example code in listing 4.7 is to create an additional static method on the `King` class, as illustrated in listing 4.8.

```

1 class StaticCallsExample {
2     void reward() {
3         Bank bank = ...
4         King.rewardPeasants(bank);
5     }
6 }
7
8 class King {
9     static King getInstance() { ... }
10
11     Set<Peasant> getPeasants() { ... }
12
13     static void rewardPeasants(Bank bank) {
14         bank.giveCoin(getInstance().getPeasants());
15     }
16 }
17
18 class Peasant { ... }
19
20 class Bank {
21     void giveCoins(Set<Peasants>) { ... }
22 }

```

Listing 4.8: Remove train wreck by creating additional static method

However, since, in our opinion, *Vignelli* should not encourage the excessive use of static utility methods, we have decided to hide occurrences of train wrecks that are caused by static method calls.

We have thus further adapted the train wreck classification condition:

$$\begin{aligned}
 & \text{isTrainWreck}(\text{expression}) \\
 & \quad = \\
 & \quad \text{type difference of expression} > 1 \\
 & \quad \vee \\
 & \quad (\text{type difference of expression} = 1 \wedge \text{length of expression} = 3 \\
 & \quad \wedge \text{expression does not contain static calls})
 \end{aligned} \tag{4.5}$$

Note that this condition does not exclude *all* method call chains that contain static calls but only those that would be considered train wrecks *because* of the static call. An example of a method call chain that *does* involve a static method call but is still rightfully classified as a train wreck is shown in listing 4.9.

```

1 King.getInstance().getLeadPeasant().getMaster();

```

Listing 4.9: Actual train wreck using static call

Here, the type difference is calculated as 2, which is the deciding factor in classifying this method call chain as a train wreck.

4.1.7 Train Wrecks in Practice: External Libraries

Using train wreck classification condition 4.5 to identify train wrecks, we have found good results in statically identifying train wrecks in source code (see evaluation section 7.1).

However, when we used this technique on our own code during testing, we encountered many examples of method call chains that were classified correctly as train wrecks, but which were not due to bad design practices on our part. This was because the code that was causing the train wrecks actually depended on external libraries whose code we were unable to modify.

In particular, when working with PSI trees (see section 3.2.1), we regularly wrote method call chains to retrieve data from the tree. One frequent example of this can be seen in listing 4.10. This example code retrieves a certain file description for a particular element in the tree.

```
1 PsiElement element = ...;  
2 element.getContainingFile().getVirtualFile();
```

Listing 4.10: Frequent train wreck in *Vignelli*

However, since `PsiElement` are defined in the IntelliJ API, there is no way for us to refactor the code to avoid this train wreck.

In fact, these spurious warnings were so prevalent in our code that we have decided to hide warnings about train wrecks that are due to the use of external libraries and which the developer is unable to remove.

To achieve this, we hide any train wreck that contains a method call whose static return type refers to a class that is not defined in the project's own source files.

4.2 Refactoring

We have devised a series of refactoring steps to remove a train wreck. Figure 4.4 shows a flow chart of these steps and the entry points (in pink) into the refactoring process.

4.2.1 Example Step-By-Step Walkthrough

To give an overview of the general steps that are required as part of this refactoring process, we will give an example walkthrough here. More detailed explanations and edge cases can be found in the following sections.

Inline Variable

If a train wreck is being assigned to a variable (details in section 4.2.2), the first step is to inline it. Consider the example code in listing 4.11 where we have found a train wreck (highlighted in pink) that is assigned to a variable.

After inlining the variable (details in section 4.2.3), the code now no longer features the `zip` variable (see listing 4.12).

Extract Method

The next step is to extract the code that is affected by the train wreck into its own method (details in section 4.2.4). As we can see in listing 4.13, we have created a new `xyz` method that

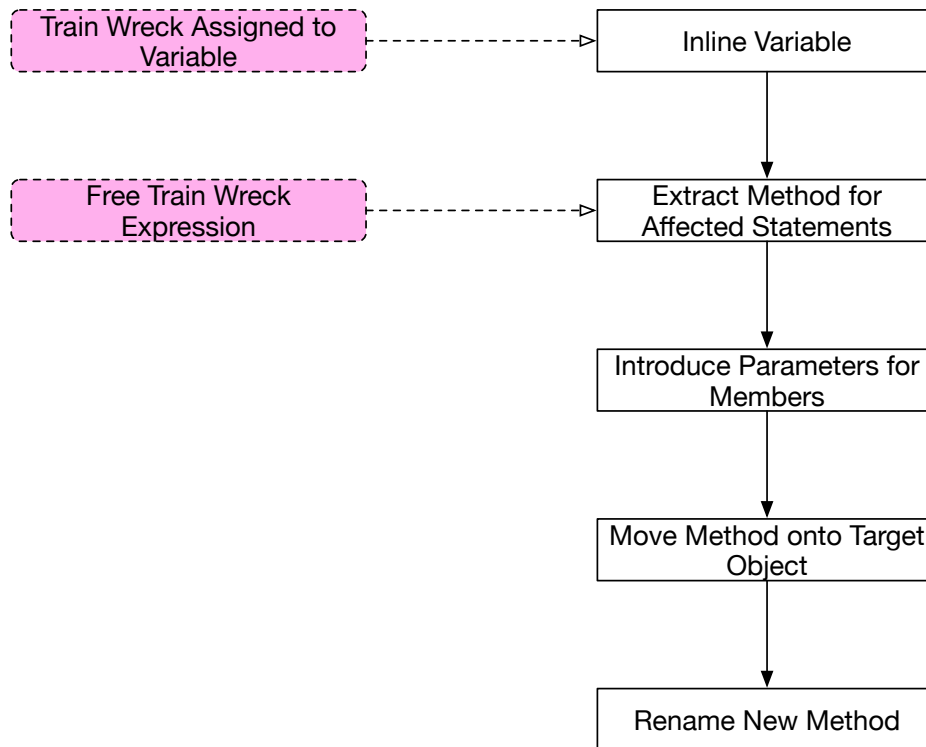


Figure 4.4: General refactoring steps to remove train wreck

```

1 public class ZipCodeExample {
2
3     private Customer customer;
4
5     void prepare() {
6         customer = new Customer();
7         ZipCode zip = customer.getAddress().getZipCode();
8         Label label = new Label();
9         label.addLine(zip.toString());
10    }
11 }
  
```

Listing 4.11: Initial code before train wreck refactoring

```

1 public class ZipCodeExample {
2
3     private Customer customer;
4
5     void prepare() {
6         customer = new Customer();
7         Label label = new Label();
8         label.addLine(customer.getAddress().getZipCode().toString());
9     }
10 }
  
```

Listing 4.12: Train wreck code after Inlining

encapsulates the train wreck statement. The name “xyz” is only an intermediate name and will be changed at a later stage when we have more context.

```

1 public class ZipCodeExample {
2
3     private Customer customer;
4
5     void prepare() {
6         customer = new Customer();
7         Label label = new Label();
8         xyz(label);
9     }
10
11     void xyz(Label label) {
12         label.addLine(customer.getAddress().getZipCode().toString());
13     }
14 }

```

Listing 4.13: Train wreck code after extracting a method for the train wreck

Introduce Parameters for Members

The next step is to introduce parameters for all fields and methods of `ZipCodeExample` in the `xyz` method (details in section 4.2.5). This is so that the method can easily be moved onto the `Customer` class in the next step without having to reference members from its old class. The resulting code after this refactoring step can be seen in listing 4.14, where we have introduced a new `Customer customer1` parameter to the `xyz()` method.

```

1 public class ZipCodeExample {
2
3     private Customer customer;
4
5     void prepare() {
6         customer = new Customer();
7         Label label = new Label();
8         xyz(label);
9     }
10
11     void xyz(Label label, Customer customer1) {
12         label.addLine(customer1.getAddress().getZipCode().toString());
13     }
14 }

```

Listing 4.14: Train wreck code after introducing parameter for customer

Move Method

The next and most significant step is to move the new `xyz()` method from its old `ZipCodeExample` class onto the `Customer` class (details section 4.2.6). We move it here because `Customer` used to be the class that was first *asked* for data. By moving the new method of type `void` onto `Customer`, `ZipCodeExample` can *tell* it what to do instead. The result of moving the method can be seen in listing 4.15. As we can see, the method chain is now smaller in `Customer::xyz()` and the code in `ZipCodeExample` no longer asks `Customer` for data.

Rename Method

The last step in the refactoring process is to rename the `xyz` method to a more sensible method name (details in section 4.2.7). After having moved the method onto its final destination class in the previous step this is now easier. For example, here we can rename the method to `fillLabel`, as shown in listing 4.16.

```

1 public class ZipCodeExample {
2
3     private Customer customer;
4
5     void prepare() {
6         customer = new Customer();
7         Label label = new Label();
8         customer.xyz(label);
9     }
10
11 }
12
13 class Customer {
14     ...
15
16     void xyz(Label label) {
17         label.addLine(getAddress().getZipCode().toString());
18     }
19 }

```

Listing 4.15: Code after moving method onto `Customer` `customer1`

```

1 public class ZipCodeExample {
2
3     private Customer customer;
4
5     void prepare() {
6         customer = new Customer();
7         Label label = new Label();
8         customer.fillLabel(label);
9     }
10
11 }

```

Listing 4.16: Final code after method rename

In the following sections, we will explain these refactoring steps in more detail and also cover the edge cases that *Vignelli* handles.

4.2.2 Refactoring Entry Points

As we can see in figure 4.4, there are two entry points to the train wreck refactoring process:

Train Wreck Assigned To Variable This entry point is used when *Vignelli* has identified a train wreck that is being assigned to a local variable (see listing 4.17). If *Vignelli* finds a train wreck that fits this description, the first refactoring step is to inline the variable to which the train wreck is assigned.

Free Train Wreck Expression This entry point is used in all other cases, when *Vignelli* has identified a train wreck. These *free* train wrecks are not bound by a variable assignment, but instead are used, for example, as arguments in other method calls (see listing 4.18) or are statements themselves (see listing 4.19). When *Vignelli* encounters a free train wreck expression, the “Inline Variable” step can be skipped and the plugin progresses to the “Extract Method for Affected Statement” refactoring step.

```
1 ZipCode zip = customer.getAddress().getZipCode();
```

Listing 4.17: Train wreck assigned to variable

```
1 void example() {  
2     label.addLabel(customer.getAddress().getZipCode());  
3 }
```

Listing 4.18: Train wreck used as a parameter to other method call

```
1 void example() {  
2     customer.getAddress().remove();  
3 }
```

Listing 4.19: Train wreck as a statement

4.2.3 Inline Variable

As described in section 4.2.2, *Vignelli* handles train wrecks that are assigned to variables (such as the one in listing 4.17) differently to others by adding an additional “Inline Variable” refactoring step.

This refactoring step involves inlining the variable that contains the train wreck into *all* statements in the code that reference said variable. For example, the code in listing 4.20 would be transformed into the code shown in listing 4.21 in this refactoring step.

```
1 ZipCode zip = customer.getAddress().getZipCode();  
2 label.addLine(zip.toString());  
3 if (!customer.isValid()) {  
4     System.out.println("Probably a wrong zip code: " + zip.toString());  
5 }
```

Listing 4.20: Multiple usage of variable containing train wreck

```
1 label.addLine(customer.getAddress().getZipCode().toString());  
2 if (!customer.isValid()) {  
3     System.out.println(  
4         "Probably a wrong zip code: " + customer.getAddress().getZipCode().toString()  
5     );  
6 }
```

Listing 4.21: Inlined train wreck variable

Refactoring Step Goal Checker

A pseudocode representation of the “Inline Variable” refactoring step goal checker can be seen in listing 4.22.

As explained in section 3.2.5, this goal checker approximates the identification of an inline refactoring.

Listing 4.22: Inline refactoring step goal checker

```
1 Before Observation:
2   affectedStatements := set of statements referencing the variable to inline.
3
4 After every PSI tree change:
5   for stmt in affectedStatements:
6     if stmt contains reference to variable:
7       return NOT_INLINED
8   return INLINED
```

4.2.4 Extract Method Refactoring Step

The second step in a train wreck variable refactoring and the first step in a free train wreck expression refactoring is the “Extract Method” refactoring step. Like its name suggests, this refactoring step extracts a sequence of statements as a new method in the same class.

For instance, consider listing 4.23 which contains two lines of code that feature a train wreck that previously used to be assigned to the variable, but was inlined using the “Inline Variable” refactoring step (see section 4.2.3). The two statements that are affected are highlighted in pink.

```
1 class ZipCodeExample {
2   private Customer customer = ...
3   private Label label = new Label();
4
5   void prepare() {
6     label.addLine(customer.getAddress().getZipCode().toString());
7     System.out.println("added label:" + customer.getAddress().getZipCode());
8     System.out.println("Done.");
9   }
10 }
```

Listing 4.23: Train wreck code with inlined variable

When the “Extract Method” refactoring is launched with these two statements marked for extraction, this step’s goal checker will watch the PSI tree for a structure that contains the two statements in an extracted method, as can be seen in figure 4.24. Here, the two statements have been extracted as a method `xyz`, which is then called from the `prepare` method. At this stage of the refactoring process, the name of the new method is not yet important. In fact it is difficult to give the new method a good name which is why we use “xyz” as a temporary solution. We will correct this at a later stage.

As usual, this refactoring can be invoked using built-in IntelliJ functionality (e.g. keyboard combination for “Extract Method” refactoring), manually by moving and writing code oneself, or by other means.

Notice that *Vignelli* will attempt to extract a new method for the *statements* that are affected by the train wreck, not the train wreck *expression*. This subtle difference results in the `void` return type of the new method that is extracted. This `void` return type expresses the intention to *tell* a neighbouring object to perform an operation instead of *asking* for data.

Of course, if the “Extract Method” refactoring step is launched as the entry point to a free train wreck expression refactoring, the set of affected statements is actually a singleton set of the statement that contains the train wreck (see listing 4.25). This single statement is thus marked for extraction and watched by the goal checker.

```

1 class ZipCodeExample {
2     private Customer customer = ...
3     private Label label = new Label();
4
5     void prepare() {
6         xyz();
7         System.out.println("Done.");
8     }
9
10    void xyz() {
11        label.addLine(customer.getAddress().getZipCode().toString());
12        System.out.println("added label:" + customer.getAddress().getZipCode());
13    }
14 }

```

Listing 4.24: Train wreck code extracted into method

```

1 void prepare() {
2     label.addLine(customer.getAddress().getZipCode().toString());
3 }

```

Listing 4.25: Free train wreck expression: statement marked for extraction

Handling Nested Statements

When inlining a local variable that contains a train wreck, it is frequently the case that the variable is referenced multiple times in the code. This causes the train wreck to duplicate multiple times. Consider again the result of inlining `zip` from listing 4.20. Listing 4.26 again lists the results of this inline operation, however, this time highlighting the particular statements that were affected.

```

1 label.addLine(customer.getAddress().getZipCode().toString());
2 if (!customer.isValid()) {
3     System.out.println("Probably a wrong zip code: " + customer.getAddress().getZipCode().toString());
4 }

```

Listing 4.26: Affected statements after inlining `zip`

As we can see in the code, the inlining of variable has affected two statements, each at a different block depth; while the first affected statement (which adds to the label) is at block depth 0, the next affected statement resides inside an `if`-statement and is therefore at block depth 1.

Extracting only these two statements together, however, is impossible when trying to retain the original control flow (only execute the second statement if the condition `customer.isValid()` does not hold).

Instead, to counteract this problem, *Vignelli* will find the appropriate block of statements that retains the original control flow and can also be extracted. This is achieved by recursively including all parent statements of those affected statements that have a block depth greater than the lowest that can be found.

In this example, *Vignelli* will therefore include the `if`-statement in the list of statements marked for extraction.

Unrelated Code In-Between Statements to Extract

Consider the example in listing 4.27.

```
1 label.addLine(customer.getAddress().getZipCode().toString());
2 System.out.println("Check the customer validity");
3 if (!customer.isValid()) {
4     System.out.println("Probably a wrong zip code: " + customer.getAddress().getZipCode().toString());
5 }
6 System.out.println("Looks like it's valid");
```

Listing 4.27: Unrelated code contained within block to extract

Here, line 2 and 6 contain code that are unrelated to the train wreck. While line 6 follows the block of code that should be extracted, line 2 is located between two affected statements. IntelliJ's refactoring system already handles this case gracefully by including line 2 in the code to extract, but leaving line 6 behind. In *Vignelli*, we have experimented with this feature and have come to the conclusion that this approach also leads to sensible extractions in the context of the train wreck refactoring.

Refactoring Step Goal Checker

Listing 4.28 contains pseudocode for the refactoring step goal checker used to verify successful extraction of the given statements.

Listing 4.28: Extract method refactoring step goal checker

```
1 Before Observation:
2   origM := set of all methods in the class that is being modified
3   callerMethod := method containing statements for extraction
4   ‘
5 After every PSI tree change:
6   newM := set of all methods in the class that is being modified
7   addedM := newM – origM
8   for method in addedM:
9       if method contains block of statements up for extraction:
10          if callerMethod contains call to method:
11              return EXTRACTED
12   return NOT_EXTRACTED
```

As we can see, the code checks newly added methods in the class being modified for code that resembles the block of statements that were marked to be extracted. If such a method exists and a call to this new method has been inserted in the old method that used to contain the statements, the goal checker reports a successful extraction.

4.2.5 Introduce Parameters for Members

After the code affected by the train wreck has been extracted into a new method, the next step in the refactoring process is to introduce parameters for all references to member variables or methods.

To understand why, refer back to the general refactoring steps required to eradicate the train wreck, displayed in figure 4.4. As we can see, the newly-extracted method will be moved to another class in a later step. More specifically, after this particular “Introduce Parameter” step, the method containing the train wreck will be moved onto the class of the neighbouring

object in the object navigation structure that is currently being asked for data. In the example depicted in listing 4.24, this neighbouring object is `customer`, which means that the method will be moved onto the `Customer` class. A more detailed explanation of this choice is given in section 4.2.6.

Notice, however, that the method to move `void xyz(){ ... }` still contains references to `label`, which is a private instance variable defined in the `ZipCodeExample` class. If we were to move the method as it is onto `Customer customer`, we would be unable to reference this variable. Even if we were to pass the current `ZipCodeExample` instance to the moved method, we would be unable to access the `label` due to its `private` visibility.

The solution to this problem is therefore to avoid having to reference the instance variable altogether in the method that is scheduled to be moved. In this particular example, this means that we are required to remove all direct references to `ZipCodeExample`'s `label` member variable.

The best way to achieve this goal is to introduce a parameter for the `label` field to the `void xyz(){ ... }` method. Listing 4.29 shows how this can be done: instead of referring to `label` directly in the `xyz()` method, it is passed as an argument. The result of this operation is that inside of `xyz(Label theLabel)` the code no longer refers to the `label` field.

```

1 class ZipCodeExample {
2     private Customer customer = ...
3     private Label label = new Label();
4
5     void prepare () {
6         xyz (label );
7         System.out.println("Done.");
8     }
9
10    void xyz (Label theLabel) {
11        theLabel.addLine(customer.getAddress().getZipCode().toString());
12        System.out.println("added label:" + customer.getAddress().getZipCode());
13    }
14 }

```

Listing 4.29: Introducing a parameter for label member variable

More generally, it can be observed that this problem arises for any reference to an instance variable or another member method that is located in the same class. For this reason, the “Introduce Parameter” refactoring step actually ensures that parameters are introduced for *all* references to members of the same object whose class contains the method to move.

In practice, this means that the “Introduce Parameter” refactoring step consists of as many steps as there are references to member variables or methods (see figure 4.5).

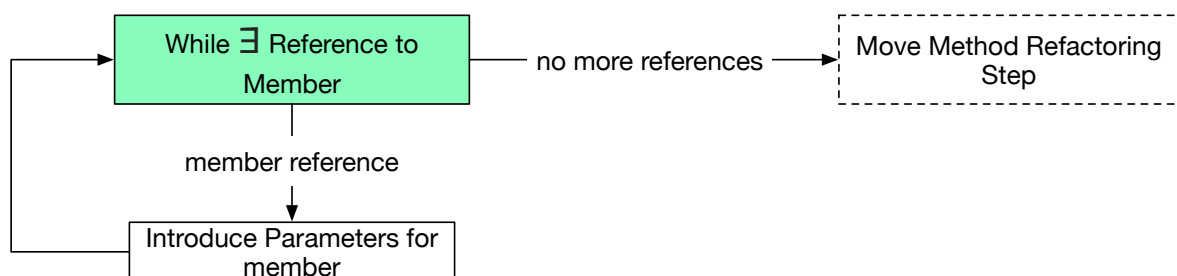


Figure 4.5: Parameters are introduced for *all* member references

Note that since we are also handling member methods, nesting of references to members can occur, as shown by the example in listing 4.30.

```

1 class NestedMemberReferencesExample {
2     void xyz() {
3         ...
4         prepareCustomer(customer);
5         ...
6     }
7
8     private Customer prepareCustomer(Customer customer) {
9         ...
10    }
11 }

```

Listing 4.30: Nested references to members

At the moment, *Vignelli* processes the member references in an undefined order. This means that in the example, it could be the case that the developer first introduces a parameter for `customer`, only to follow it by introducing a parameter for the containing expression `prepareCustomer(customerParam)`.

However, if the developer following the *Vignelli* instructions uses IntelliJ’s built-in refactoring tools as suggested by the tool, the IDE will ensure that the old `customerParam` parameter is subsequently removed after the introduction of a parameter for `prepareCustomer(customer)`. In future versions, we plan to reduce friction in this step by only suggesting the introduction of parameters for the outermost member references.

Introduce Parameters Refactoring Step Goal Checker

The “Introduce Parameters for Members” refactoring step is special as this step actually consists of multiple single parameter introductions.

Overall, the “Introduce parameters for Members” refactoring step reports a successful completion of the step to the corresponding refactoring when no more member variables or methods are referenced in the method that is scheduled to be moved.

Note that this definition of the goal also covers the case in which *no* members are referenced in the method to move — in this case the step immediately reports completion upon starting to watch for changes and the refactoring will automatically move onto the next step: “Move method”.

Alternatively, if members are referenced in the method, the refactoring step in turn creates internal refactoring steps that cover the introduction of one parameter for a given expression. Each of these internal refactoring steps watches the PSI tree for changes and uses the algorithm in listing 4.31 to determine whether a parameter has successfully been introduced.

As we can see, the goal checker considers a parameter introduction for a given expression to be successful when said expression is passed as an argument in all calls to the method.

4.2.6 Move Method Refactoring Step

The next refactoring step in the *Vignelli* refactoring process to remove a train wreck is to move the newly-created helper method to another class.

To understand where the method should be moved, consider again the example in listing 4.29. In its current state, the helper method (`xyz`) still contains the train wrecks with which the refactoring process was initially started. As discussed in section 2.4.1, the cause of this train wreck is that `ZipCodeExample` is *asking* `Customer` for data that is needed to perform a certain operation (adding the zip code to the label). Instead, `ZipCodeExample` could also *tell*

Listing 4.31: Introduce a single parameter refactoring step goal checker

```

1 Before Observation:
2   origExpression := the expression for which a parameter is introduced
3   origP := set of all parameters of the method
4
5 After every PSI tree change:
6   newP := set of all parameters of the method
7   addedP := newP - origP
8   for param in addedP:
9     calls = set of all calls to the method
10
11     propagated := True
12     for call in calls:
13       if not (call passes argument for param  $\wedge$  argument for param ==
14             origExpression):
15         propagated := False
16
17     if propagated:
18       return INTRODUCED
19
20 return NOT INTRODUCED

```

the `Customer` to perform this operation directly. To achieve this, the operation to be performed on the object that was previously *asked* for data. In this example, this is `Customer customer`.

We therefore move the `xyz` method onto the `Customer` class and call it on the `customer` object. The resulting code can be seen in listing 4.32.

```

1 class ZipCodeExample {
2   private Customer customer = ...
3   private Label label = new Label();
4
5   void prepare() {
6     customer.xyz (label);
7     System.out.println("Done.");
8   }
9
10 }
11
12 class Customer {
13   ...
14
15   void xyz (Label theLabel) {
16     theLabel.addLine(getAddress().getZipCode().toString());
17     System.out.println("added label:" + getAddress().getZipCode());
18   }
19 }
20 }

```

Listing 4.32: Moving method to neighbour to avoid asking for data

Examining the code in the `Customer::xyz()` method, we can see that the length of the train wreck has been reduced by 1.

In general, we therefore aim to move the helper method containing the train wreck onto the neighbour object that we previously asked for data.

Move Method Refactoring Step Goal Checker

When the “Move Method” refactoring step is instantiated, *Vignelli* also passes the desired target to the step’s constructor.

The target class is computed by the step’s corresponding refactoring instance. In most cases, this is done by inspecting the method code and finding train wrecks using the `TrainWreck-IdentificationEngine` and picking the neighbour instance which is first asked for data. Cases in which this approach does not work are discussed in section 4.3.

The target class is later used in the refactoring step goal checker to identify whether the method has successfully been moved to the correct class. Pseudocode for how the goal checker does this can be seen in listing 4.33.

Listing 4.33: Move method refactoring step goal checker

```

1 Before Observation:
2   origM := set of all methods in the target class
3
4 After every PSI tree change:
5   newM := set of all methods in the target class
6   addedM := newM - origM
7   for method in addedM:
8     if addedM contains all statements from original method:
9       return METHOD.MOVED
10  return METHOD.NOT.MOVED

```

As we can see, the goal checker watches the target class’ PSI tree for new methods that contain all the statements that are contained in the method to move.

4.2.7 Rename Method Refactoring Step

The last refactoring step in the refactoring process to eradicate a train wreck is the “Rename Method“ refactoring step. As its name suggests, this step involves the renaming of the new method that was successfully moved onto the caller’s neighbouring object.

Even though this step is not strictly required when trying to remove train wrecks, naming variables and methods is one of the most important skills to have as a software engineer and using good names for variables and methods is important [5, ch. 2].

For this reason, we include the “Rename Method“ refactoring step here as it requires the developer to think carefully about an appropriate name for the newly-moved method. Notice also, that for inexperienced developers it may be difficult to name the method immediately after extracting it in the “Extract Method” refactoring step (see section 4.2.4) as it may not be clear in which context the method will later be called. However, having moved the method onto its target class it is now easier to describe the method.

“Rename Method” Refactoring Step Goal Checker

The “Rename Method” refactoring step goal checker is a special case of a goal checker as IntelliJ provides notifications when methods are renamed. *Vignelli* uses this functionality in the goal checker and listens for these `RENAME` events.

In its current state, it is not possible to skip this refactoring step as a user and leaving the name unmodified will not allow the developer to progress to the next screen (a short review of all the steps in this refactoring process). However, this is only a limitation of the current user interface and can easily be fixed in future versions of the plugin.

4.3 Refactoring Train Wrecks Involving Fluent Interfaces

In section 4.2 we have discussed the general refactoring steps that are required to remove train wrecks.

Although these steps remain valid in the context of train wrecks involving fluent interfaces, there are some special considerations that we need to make.

To discuss these differences, consider the code in listing 4.34. From table 4.1 we know that this code contains a train wreck, despite the fact that it uses the builder pattern. This is because of the additional call to `getAttempts()` after the building of the `Request` instance which is subsequently asked for the number of remaining attempts.

```
1 public class BuilderExample {
2     public void execute() {
3         Request.Builder builder = new Request.Builder();
4         PacketManager manager = new PacketManager();
5         manager.recordRemainingAttempts(
6             builder.withUrl("http://example.com").withContent("")
7                 .withAttempts(10).build().getAttempts()
8         );
9     }
10 }
11
12 class PacketManager {
13     public void recordRemainingAttempts(int attempts) {
14         ...
15     }
16 }
```

Listing 4.34: Train wreck in method chain involving fluent interface via builder pattern

To understand why the normal refactoring steps described in section 4.2 do not lead to the best results in this case, consider how the code would change from step to step. In particular, listing 4.35 shows the state of the code after extracting a method for the statements affected by the train wreck.

```
1 public class BuilderExample {
2     public void execute() {
3         PacketManager manager = new PacketManager();
4         Request.Builder b = new Request.Builder();
5         xyz(manager, b);
6     }
7
8     void xyz(PacketManager manager, Request.Builder b) {
9         manager.recordRemainingAttempts(
10            b.withUrl("http://example.com").withContent("")
11                .withAttempts(10).build().getAttempts()
12        );
13    }
14 }
15
16 class PacketManager {
17     public void recordRemainingAttempts(int attempts) {
18         ...
19     }
20 }
```

Listing 4.35: Code state after method extraction for train wreck using normal refactoring steps

To prepare the next step (“Move Method”, see section 4.2.6), the corresponding **Refactoring** will attempt to find the neighbouring object that is first asked for data and therefore the begin-

ning of the train wreck. In the example, this is the `Request.Builder` instance `b` (highlighted in pink).

Having established that *Vignelli* would choose the `Request.Builder` class as a target class for the “Move Method” operation, consider now the effects performing this step. The resulting code after moving the method to the computed target class, `Request.Builder`, can be seen in listing 4.36.

```
1 public class BuilderExample {
2     public void execute() {
3         PackageManager manager = new PackageManager();
4         Request.Builder b = new Request.Builder();
5         b.xyz(manager);
6     }
7 }
8
9 class Request {
10     ...
11     static class Builder {
12         ...
13
14         void xyz(PackageManager manager) {
15             manager.recordRemainingAttempts(
16                 .withUrl("http://example.com").withContent("")
17                 .withAttempts(10).build().getAttempts()
18             );
19         }
20     }
21 }
22
23 class PackageManager {
24     public void recordRemainingAttempts(int attempts) {
25         ...
26     }
27 }
```

Listing 4.36: Code state after moving method to immediate neighbour using normal refactoring steps

Although the calling code now *tells* its immediate neighbour to perform a certain logging operation, semantically the resulting code no longer makes sense: a builder is told to perform a logging operation on the object that it is building. In addition, the method that is now called on the builder, `xyz()` can *only* record the remaining attempts of a `Request` instance that was constructed in one specific way.

Overall, despite our best intentions, we therefore seem to have worsened the design of the code. Clearly, choosing the immediate neighbour object that is first being asked to compute a value as a target for the new method is not an appropriate choice in the context of fluent interfaces.

4.3.1 Critical Call Chains

To avoid the aforementioned problems, *Vignelli* treats train wrecks that involve method chains using fluent interfaces differently from others.

In particular, the plugin uses the concept of a “critical call chain” which we define as follows:

Definition 3. *A critical call chain is the longest method call chain contained within a given train wreck, which should not be considered a train wreck, starting at the same method call qualifier.*

A critical call therefore consists of the train wreck with as many method calls at the end of the chain removed as is required in order for the resulting method call chain not to be classified as a train wreck. Table 4.2 contains examples of train wrecks with their corresponding critical call chains.

Train Wreck	Static Types	Critical Call Chain
<code>builder.withUrl("http://example.com").withContent("").withAttempts(10).build().getAttempts()</code>	A:A:A:A:B:C	<code>builder.withUrl("http://example.com").withContent("").withAttempts(10).build()</code>
<code>customer.getAddress().getZipCode()</code>	A:B:C	<code>customer.getAddress()</code>

Table 4.2: Critical call chain examples

In order to find the critical call chains of train wrecks, *Vignelli* uses the algorithm depicted in listing 4.37.

Listing 4.37: Algorithm to find the critical call chain of a given train Wreck

```

1 criticalCallChain(chain):
2     typeDifference = typeDifference(chain)
3     while isTrainWreck(typeDifference, length of chain):
4         chain = chain with last call removed
5     return chain

```

4.3.2 Additional Step: Introduce Parameter for Critical Call Chain

In *Vignelli* we use the concept of *critical call chains* to handle the train wreck refactoring process differently when fluent interfaces are involved.

However, since this additional step is *only* required when dealing with fluent interfaces, we have had to devise a condition under which we perform additional refactoring steps. As it turns out the following simple condition has given good results:

$$\text{length of critical call chain} > 2 \tag{4.6}$$

When this condition holds, *Vignelli* performs an additional refactoring step: “Introduce Parameter for Critical Call Chain”. This refactoring step is performed directly after the extraction of the train-wreck code into its own method (see figure 4.6).

In the “Introduce Parameter for Critical Call Chain” refactoring step, as the name suggests, a parameter is introduced for the expression that is the critical call chain of the train wreck.

To illustrate this, consider again the state of the code during the refactoring after the extraction of a method in listing 4.35.

As we know from table 4.2, the critical call chain of the train wreck is `builder.withUrl("http://example.com").withContent("").withAttempts(10).build()`. Clearly, the length of this chain is greater than 2, which means that *Vignelli* will indeed launch the “Introduce Parameter for Critical Call Chain” refactoring step.

Similarly to the introduction of parameters for member variables and methods, the plugin now launches an “IntroduceParameterRefactoringStep”, only now pointing to the critical call

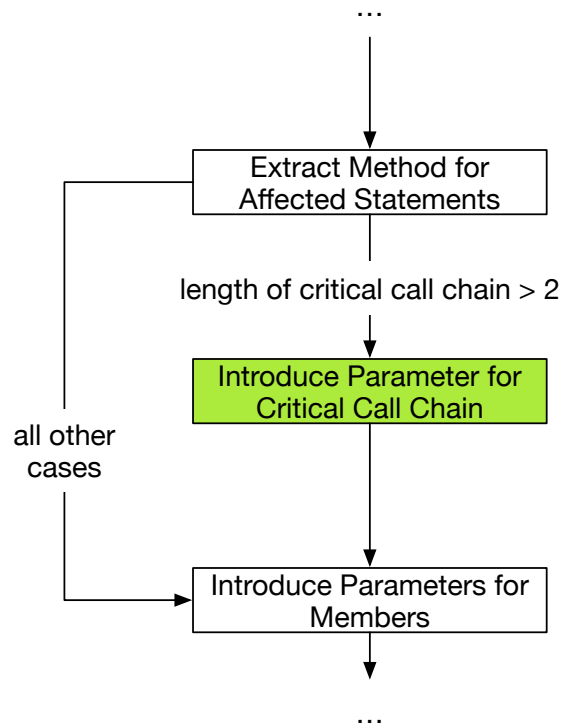


Figure 4.6: A parameter for the critical call chain is introduced when required

chain for the expression to introduce. Performing this step results in the code that can be seen in listing 4.38.

```

1 public class BuilderExample {
2     public void execute() {
3         Request.Builder builder = new Request.Builder();
4         PacketManager manager = new PacketManager();
5         xyz(manager, b.withUrl("http://example.com").withContent("").withAttempts(10).build() );
6     }
7
8     void xyz(PacketManager manager, Request request) {
9         manager.recordRemainingAttempts(request.getAttempts());
10    }
11 }
12
13 class PacketManager {
14     public void recordRemainingAttempts(int attempts) {
15         ...
16     }
17 }
  
```

Listing 4.38: Code state after introducing a parameter for the critical call chain

As we can see in this example, the fully-constructed request is passed directly as an argument to the helper method. This means that this call chain will remain in the `execute()` method in the next refactoring step when `xyz` is moved. As a move target, *Vignelli* now again selects the closest neighbour that is asked for data in the helper method: `Request`.

Performing the “Move Method” refactoring step will subsequently result in the code shown in listing 4.39.

As we can see, the highlighted method chain consists of the construction of the request,

```

1 public class BuilderExample {
2     public void execute() {
3         PackageManager manager = new PackageManager();
4         Request.Builder b = new Request.Builder();
5         b.withUrl("http://example.com").withContent("").withAttempts(10).build().xyz(manager);
6     }
7
8 }
9
10 class Request {
11     ...
12     void xyz(PackageManager manager) {
13         manager.addDataPoint(getAttempts());
14     }
15 }
16
17 class PackageManager {
18     public void recordRemainingAttempts(int attempts) {
19         ...
20     }
21 }

```

Listing 4.39: Code state after moving helper method, leaving critical call chain behind

followed by a method call of type `void` (see section 4.1.5). According to our classification condition, this method chain will not be identified as a train wreck — we have successfully refactored code containing a train wreck that involves the use of fluent interfaces.

Chapter 5

Identification and Eradication of Direct Singleton Retrievals

A second problem that *Vignelli* is able to identify and eliminate through suggested refactoring steps is that of a “direct use of a singleton”, discussed in section 2.4.2. This chapter discusses the techniques that we use to realise this.

5.1 Identification

Similarly to the identification of train wrecks, rather than using a metrics-based approach, we have chosen to identify the direct use of a singleton by investigating the structure of classes and the interactions between them.

Since *Vignelli* only analyses the code of the file that is currently being edited by the user (for speed reasons), we have chosen the following approach to identifying direct uses of singleton classes:

1. Find all potential instance retrieval methods. In *Vignelli*, we use the set of all static calls in the method that is being analysed.
2. Classify the static calls’ corresponding classes as either singletons or non-singletons. This is done by analysing the structure of the class as opposed to using a metrics-base approach, because the singleton pattern can be described well in terms of its structure and usage, as illustrated by the simple UML diagram, reproduced here in figure 5.1.

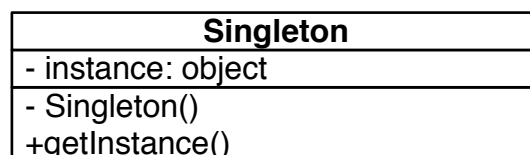


Figure 5.1: UML for singleton pattern

Based on the information that can be extrapolated from this diagram, *Vignelli* therefore checks that:

- the instance retrieval method candidate’s (the static call under consideration) name is “getInstance”,

- the instance retrieval method candidate takes no arguments, and
- the class features only private constructors.

Figure 5.2 shows how the identification procedure filters out static method calls that do not fit the generic description of a singleton.

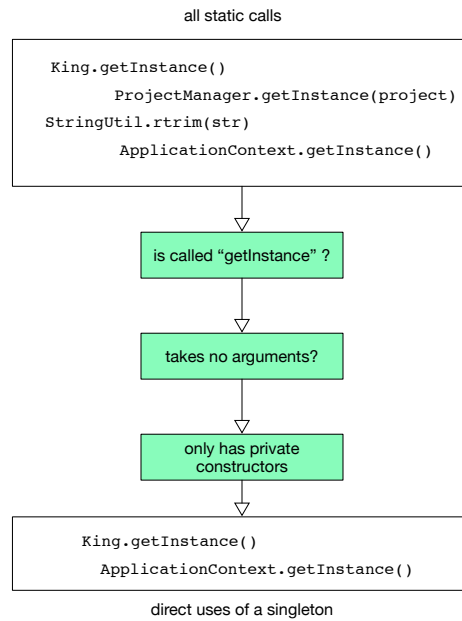


Figure 5.2: Identification of direct uses of singletons

5.2 Refactoring

The refactoring steps that we have devised in order to remove any uses of a singleton inside a particular class can be seen in figure 5.3. The refactoring steps are designed to remove the direct references to the singleton class and instead inject an instance of an interface that is implemented by the singleton into the class constructor (i.e. dependency injection).

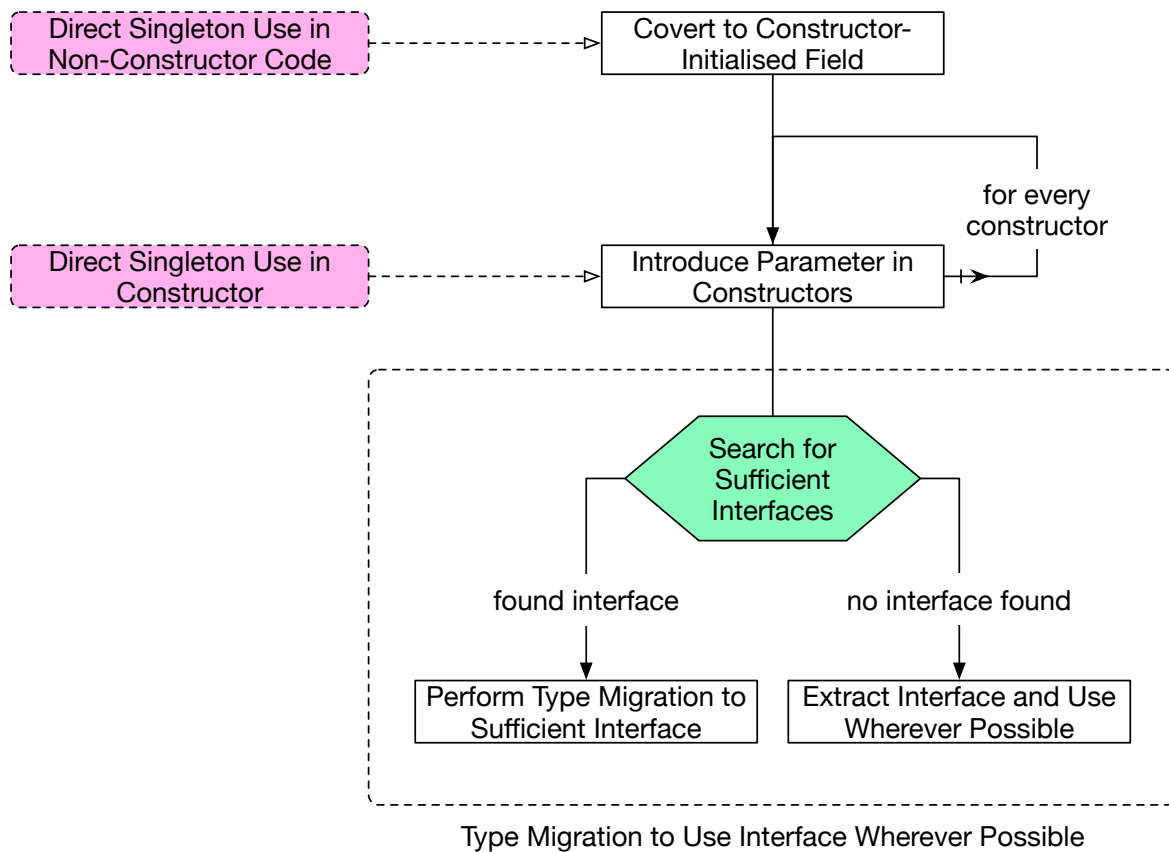


Figure 5.3: General refactoring steps to remove direct use of singleton

5.2.1 Example Step-By-Step Walkthrough

To give an example of what refactoring steps are generally required when refactoring towards dependency injection instead of direct use of singletons, we give an example walkthrough here. More detail on each refactoring step can be found in the following sections.

Convert to Constructor-Initialised Field

If the direct use of a singleton can be found in a non-constructor method (details in section 5.2.2), the first step in the refactoring process is to convert the instance retrieval expression to a field that is initialised in the constructor (details in section 5.2.3). Consider listing 5.1 in which we can find a singleton instance retrieval method call (highlighted in pink) in the `send()` method.

```

1 class Mailer {
2     public void send(String address, String body) {
3         String fromAddress =.AppSettings.getInstance().getMailFromAddress();
4     }
5 }
  
```

Listing 5.1: Direct use of singleton before refactoring

Converting this expression to a field that is initialised in the constructor of the class results in the code that can be seen in listing 5.2.

```

1 class Mailer {
2     private final AppSettings settings;
3
4     public Mailer() {
5         settings = AppSettings.getInstance();
6     }
7     public void send(String address, String body) {
8         String fromAddress = settings.getMailFromAddress();
9     }
10 }

```

Listing 5.2: Singleton instance retrieval after conversion to constructor-initialised field

Introduce Parameter in Constructor

The next step in the refactoring process is to inject the `AppSettings` dependency as a parameter to the constructor (details in section 5.2.4). Doing so results in the code that can be seen in listing 5.3. As we can see, users of the `Mailer` class now inject the `AppSettings` instance into `Mailer`, which no longer contains any instance retrieval method calls.

```

1 class MailerUser {
2     public void mailUsers() {
3         Mailer mailer = new Mailer( AppSettings.getInstance() );
4         ...
5     }
6 }
7
8 class Mailer {
9     private final AppSettings settings;
10
11     public Mailer(AppSettings theSettings) {
12         settings = theSettings;
13     }
14     public void send(String address, String body) {
15         String fromAddress = settings.getMailFromAddress();
16     }
17 }

```

Listing 5.3: Injecting the `AppSettings` dependency

Type Migration to Use Interface Whenever Possible

The final step in the refactoring process is to use an interface instead of the `AppSettings` class in `Mailer` (details in section 5.2.5). This is to reduce coupling between the classes further so that `Mailer` only requires the use of a `Settings` interface rather than the specific `AppSettings` class.

Since `AppSettings` does not yet implement any existing interfaces (see discussion in section 5.2.6), we can extract a new interface and use that instead of the `AppSettings` type in the `Mailer` class. The result of performing this type migration can be seen in listing 5.4. As we can see, `Mailer` no longer refers to `AppSettings`, which now implements the new `Settings` interface.

In the following sections, we will discuss these refactoring steps in more detail and cover some of the edge cases that *Vignelli* handles.

```

1 class Mailer {
2     private final Settings settings;
3
4     public Mailer( Settings theSettings) {
5         settings = theSettings;
6     }
7     public void send(String address, String body) {
8         String fromAddress = settings.getMailFromAddress();
9     }
10 }
11
12 class AppSettings implements Settings {
13     ...
14     @Override
15     public String getMailFromAddress() { ... }
16 }
17
18 interface Settings {
19     String getMailFromAddress();
20 }

```

Listing 5.4: Using interface instead of `AppSettings`

5.2.2 Refactoring Entry Points

As we can see in figure 5.3 there are two entry points to the refactoring process (highlighted in pink):

Direct Singleton Use in Non-Constructor Code This entry point to the refactoring process is used when *Vignelli* identifies a direct singleton use in any method that is *not* a constructor of the class.

For example, listing 5.5 contains two occurrences of a direct singleton use on lines 3 and 7. The occurrence on line 7 does not appear in a constructor and therefore qualifies for this entry point.

Direct Singleton Use in Constructor This entry point to the refactoring process is used when an instance retrieval call is identified inside a constructor, such as line 3 in listing 5.5. In this case, *Vignelli* starts with the “Introduce Parameter in Constructors” step.

```

1 class UsageExample {
2     public UsageExample() {
3         AppSettings.getInstance().getMailFromAddress();
4     }
5
6     public otherMethod() {
7         AppSettings.getInstance().getMailFromAddress();
8     }
9 }

```

Listing 5.5: Different entry points for two direct singleton uses in one class

5.2.3 Convert to Constructor-Initialised Field

Since the goal of the refactoring process is to use dependency injection to inject an instance of the singleton into the object, it is the constructor’s task to ensure that the injected instance can be used within the class.

Therefore, if the singleton use is located outside of a constructor, the first step in the refactoring process is to assign the singleton retrieval expression to a field that is initialised in the constructor. The code that previously called `getInstance()` must now reference the new field instead. The example code in listing 5.6 uses the `AppSettings` singleton class (see figure 2.9 in section 2.4.2).

```
1 class Mailer {
2     public void send(String address, String body) {
3         String fromAddress = AppSettings.getInstance().getMailFromAddress();
4     }
5 }
```

Listing 5.6: Direct use of singleton in non-constructor method

The `send()` method, which is not a constructor, contains the instance retrieval method call (highlighted in pink). Since it is our goal to replace this retrieval with the use of an injected `AppSettings` instance, we convert this expression into a field and initialise it in the constructor. Listing 5.7 shows the result of performing this refactoring step.

```
1 class Mailer {
2     private final AppSettings settings;
3
4     public Mailer() {
5         settings = AppSettings.getInstance();
6     }
7
8     public void send(String address, String body) {
9         String fromAddress = settings.getMailFromAddress();
10    }
11 }
```

Listing 5.7: Direct use of singleton after conversion to constructor-initialised field

As we can see, the instance retrieval method call has been moved to the constructor and the previous occurrence has been replaced with a reference to the field (highlighted in pink). When there are multiple constructors, the new field assignment must be added in *all* constructors, as illustrated in listing 5.8;

```
1 class Mailer {
2     public Mailer() {
3         settings = AppSettings.getInstance();
4     }
5
6     public Mailer(Configuration config) {
7         settings = AppSettings.getInstance();
8     }
9
10    public void send(String address, String body) {
11        String fromAddress = settings.getMailFromAddress();
12    }
13 }
```

Listing 5.8: Mailer class with multiple constructors after introduction of constructor-initialised Field

Refactoring Step Goal Checker

Listing 5.9 shows pseudocode for the refactoring step goal checker that is used to verify the successful conversion of the instance retrieval method call expression into a field which is initialised in the constructors of the class.

Listing 5.9: Convert to constructor-initialised field refactoring step goal checker

```
1 Before Observation:
2   origExpression := expression to be converted into field
3   origExpressionMethod := method in which the expression to be converted was
   originally located
4   origAssignments := set of all assignments in constructor.
5
6 After every PSI tree change:
7   curAssignments := set of all assignments in constructor.
8   newAssignments := newAssignments - origAssignments
9
10  coveredConstructors := {}
11
12  for ass in newAssignments:
13    if rhs(ass) contains origExpression
14      && lhs(ass) is field
15      && lhs(ass) referenced in origExpressionMethod:
16        coveredConstructors <- constructor that contains ass
17
18  if coveredConstructors contains all constructors:
19    return CONVERTED
20  else:
21    return NOT.CONVERTED
```

The goal checker checks that the expression to be converted into a field is the RHS of an assignment. This assignment must be in all constructors. In addition, the LHS of said assignments must be fields which are references in the original method of the expression that was to be converted.

Handling Multiple Singleton Instance Retrievals in the Same Class

Consider listing 5.10. Here, the class contains multiple occurrences of the same instance retrieval call.

```
1 class Mailer {
2   public void send(String address, String body) {
3     String fromAddress = AppSettings.getInstance().getMailFromAddress();
4     String adminAddress = AppSettings.getInstance().getAdminAddress();
5   }
6 }
```

Listing 5.10: Multiple singleton instance retrieval calls in the same class

Ideally, if there are multiple calls to the same instance retrieval method in the same class, the resulting code should still only inject one instance of this singleton.

IntelliJ's implementation of the "Convert to Field" refactoring step already checks for multiple occurrences of the selected expression and allows users to "replace all occurrences" if they so desire (see figure 5.4).

Vignelli therefore relies on IntelliJ to let users choose this option. However, the option is highlighted and recommended in the refactoring step's explanation (see figure 5.5).

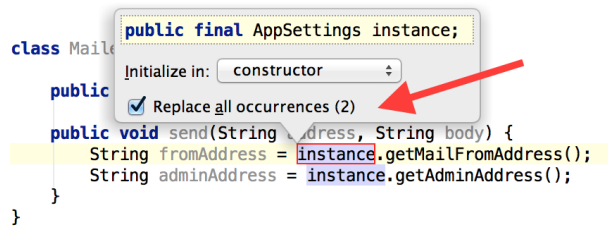


Figure 5.4: Replacing all occurrences of the instance retrieval with one field

In the following step, ensure that you:

- Select the constructor to be the initialisation place for the field.
- Choose to replace all occurrences of `AppSettings.getInstance()` in the class with the field.
- Declare the resulting field `final` (good practice)

Figure 5.5: Refactoring step explanation features note to replace all occurrences

5.2.4 Introduce Parameter in Constructors

The “Introduce Parameter in Constructors” refactoring step is the second step in the refactoring process if the singleton retrieval was located in a non-constructor method (see section 5.2.3) and the first refactoring step if the instance retrieval was already located in a constructor.

Since it is possible that multiple constructors were affected by the conversion of an instance retrieval to a constructor-initialised field, the “Introduce Parameter in Constructors” step actually consists of multiple steps: the introduction of a parameter in every constructor.

Single Parameter Introduction in One Constructor

Each single introduction of a parameter aims to eliminate the instance retrieval call occurrence in the constructor by injecting the call into the constructor as an argument. To illustrate the effects, consider again listing 5.7. Introducing a parameter will result in the new code in listing 5.11.

```

1 class MailerUser {
2     void mailUsers() {
3         Mailer mailer = new Mailer( AppSettings.getInstance() );
4         ...
5     }
6 }
7
8 class Mailer {
9     private final AppSettings settings;
10
11     public Mailer( AppSettings settings ) {
12         this.settings = settings;
13     }
14
15     public void send(String address, String body) {
16         String fromAddress = settings.getMailFromAddress();
17     }
18 }

```

Listing 5.11: Introduced constructor parameter for instance retrieval expression

We can observe that the instance retrieval method call has been moved into the `MailerUser`

class — the `Mailer` class is now free of instance retrieval methods and therefore no longer explicitly coupled to *one specific instance* of the `AppSettings` class.

The mechanics of the “Introduce Constructor Parameter” step are the same as those discussed in the “Introduce Parameter” step that was part of the train wreck refactoring process (see section 4.2.5). However, the tool window’s refactoring step explanation is customised for the singleton refactoring process.

Once all constructors have been refactored, the plugin moves onto the next step: “Type Migration”.

5.2.5 Type Migration to Use Interface Wherever Possible

After the application of the “Introduce Parameter in Constructors” refactoring step (see section 5.2.4), we can see that the `Mailer` class is no longer tightly coupled to the one and only instance of the `AppSettings` class. Although it is still tied to the class, it is possible that the `AppSettings` class will be changed in the future to no longer use the singleton pattern — in this case the `Mailer` class will not be changed. We have therefore already improved the design. However, as discussed in section 2.4.2, we can go one step further and reduce coupling by using an interface in the `Mailer` class which is implemented by the `AppSettings` class. This refactoring step, “Type Migration to Use Interface Wherever Possible”, aims to achieve this.

5.2.6 Finding Existing Interfaces

As we can see in figure 5.3, this refactoring step involves a choice between two separate steps, depending on whether any existing interfaces can be reused or not. For example, consider the `AppSettings` implementation in listing 5.12 which implements no interfaces and inherits from no base classes¹.

```
1 public class AppSettings {
2     private static final AppSettings INSTANCE = new AppSettings();
3
4     private AppSettings() { }
5
6     public static AppSettings getInstance() {
7         return INSTANCE;
8     }
9
10    String getMailFromAddress() { ... }
11 }
```

Listing 5.12: Singleton implementing no interface

Since no interfaces or base classes exist that *Vignelli* could use instead of the `AppSettings` class, the plugin will instruct the developer to “Extract [a new] Interface and Use it Wherever Possible” (see section 5.2.8).

Listing 5.13 shows a counterexample in which the `AppSettings` class implements the `Settings` interface and therefore overrides the `getMailFromAddress()` method.

To determine whether *Vignelli* can instruct the developer to use the `Settings` interface inside the `Mailer` class, the plugin considers the calling code inside the `Mailer` class.

The `Mailer` implementation in listing 5.11 only uses the `getMailFromAddress()` method of the `AppSettings` class. As we can see, this method is declared in the `Settings` interface —

¹We treat the phrase “inherits from no base classes” to mean that the class inherits from no base classes other than Java’s `Object` class

```

1 public class AppSettings implements Settings {
2     private static final AppSettings INSTANCE = new AppSettings();
3
4     private AppSettings() { }
5
6     public static AppSettings getInstance() {
7         return INSTANCE;
8     }
9
10    @Override
11    String getMailFromAddress() { ... }
12
13    String getAdminAddress() { ... }
14 }
15
16 interface Settings {
17     String getMailFromAddress();
18 }

```

Listing 5.13: Singleton implements reusable interface

we can therefore use `Settings` in place of `AppSettings`. *Vignelli* will instruct the developer to perform a type migration to use the `Settings` type in the `Mailer` implementation.

Now consider the alternative `Mailer` implementation shown in listing 5.14.

```

1 class Mailer {
2     private final AppSettings settings;
3
4     public Mailer(AppSettings settings) {
5         this.settings = settings;
6     }
7
8     public void send(String address, String body) {
9         String fromAddress = settings.getMailFromAddress();
10        String adminAddress = settings.getAdminAddress();
11        ...
12    }
13 }

```

Listing 5.14: Mailer uses all methods in `AppSettings`

In this implementation, the `Mailer` uses both the `getMailFromAddress()` and the `getAdminAddress()` that `AppSettings` implements. However, the `Settings` interface defined in listing 5.13 only defines the `getMailFromAddress()` method. The `Settings` interface is therefore *insufficient* for use in the `Mailer` class as it does not define *all* of the used methods. *Vignelli* will instruct the developer to extract a new interface (see section 5.2.8).

In general, *Vignelli* decides which step to perform based on whether any existing interfaces or base classes are sufficient for use in the class under refactoring. To find all of the interfaces that may be sufficient, the algorithm depicted in listing 5.15 is used.

When the algorithm is invoked, the `classTypeToMigrate` class is typically the singleton class, while the `singletonUser` is the class that *uses* the singleton. There are exceptions to this rule, though. To illustrate this, consider the implementation of `MailerUser` in listing 5.11. In this implementation no methods of the `AppSettings` class are actually used in `MailerUser` itself. Instead, the class only passes the new instance to the newly-created `Mailer`. Applying the previous refactoring steps to that occurrence of the singleton retrieval method call will result in the code that is shown in listing 5.16. Again, the direct singleton retrieval has been removed by way of dependency injection.

However, notice that the constructor parameter already uses the `Settings` type. This is

Listing 5.15: Algorithm to find all potential type migration targets

```

1 def getMigrationTargets(singletonUser , classTypeToMigrate):
2   usedMethods := set of methods being used in singletonUser
3   targets := set of all interfaces and superclasses of the classTypeToMigrate
4
5   for target in targets:
6     for method in usedMethods:
7       if not isAppropriateTarget(target , method):
8         remove target from targets
9         break
10  return targets
11
12 def isAppropriateTarget(target , method):
13   targetMethods := all methods of target2
14   return True iff targetMethods contains method;

```

```

1 class MailerUser {
2   private final Settings settings;
3
4   public MailerUser(Settings settings) {
5     this.setting = settings;
6   }
7
8   void mailUsers() {
9     Mailer mailer = new Mailer(settings);
10    ...
11  }
12 }

```

Listing 5.16: MailerUser passes a Settings instance through to Mailer without using it

due to **Mailer** now accepting this more general type — IntelliJ’s “Convert to Field” refactoring automatically chooses the most general type. Since the **MailerUser** constructor parameter already uses an interface type, *Vignelli* should not extract an additional interface.

The “type migration target” algorithm from listing 5.15 handles this case by including the `classTypeToMigrate` in the target types that it returns, if and only if this class is an interface itself.

In the **MailerUser** example, this means that **Settings** will be included in the potential type migration targets. In this case — the class type to migrate in the current class is already included in the set of potential migration targets — *Vignelli* will skip the type migration step as it is not required and the refactoring process completes early.

5.2.7 Perform Type Migration to Sufficient Interface

When *Vignelli* was able to identify at least one interface or superclass that are sufficient for usage in the class being refactored, the plugin suggests the migration of the singleton type to one of these alternative target types. The screenshot in figure 5.6 shows how *Vignelli* suggests one of the sufficient types in the IntelliJ type migration dialog.

Migrating the type of the constructor parameter from **AppSettings** to **Settings** also changes the type of the `settings` field in listing 5.11 to the more general **Settings** type (see listing 5.17).

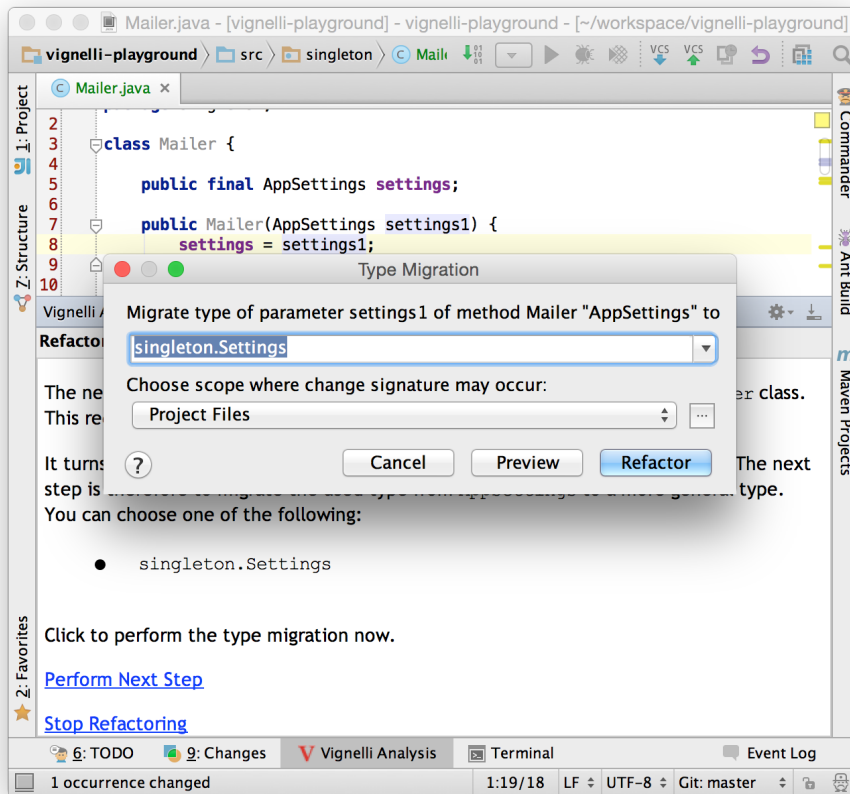


Figure 5.6: The type migration refactoring step lists all available interfaces and suggests one in the type migration dialog

```

1 class Mailer {
2     private final Settings settings;
3
4     public Mailer(Settings settings) {
5         this.settings = settings;
6     }
7
8     public void send(String address, String body) {
9         String fromAddress = settings.getMailFromAddress();
10    }
11 }

```

Listing 5.17: Mailer after type migration

Refactoring Step Goal Checker

The algorithm that is used to determine whether the type migration has been performed can be seen in listing 5.18.

This simple goal checker only checks whether the class still contains references to the original type that was to be migrated to a more general type. If this is not the case, it determines that the type migration was successful. Naturally, this goal checker has limitations, which we discuss in section 7.2.

Listing 5.18: Type migration refactoring step goal checker

```
1 Before Observation:
2   origType := the original type that should be migrated
3
4 After every PSI tree change:
5   if class contains references to origType:
6     return NOTMIGRATED
7   else:
8     return MIGRATED
```

5.2.8 Extract Interface and Use Wherever Possible

If *Vignelli* is unable to find any existing interfaces that can be used in place of the singleton class, the last step of the refactoring process is to extract a new interface and use it wherever possible in the class.

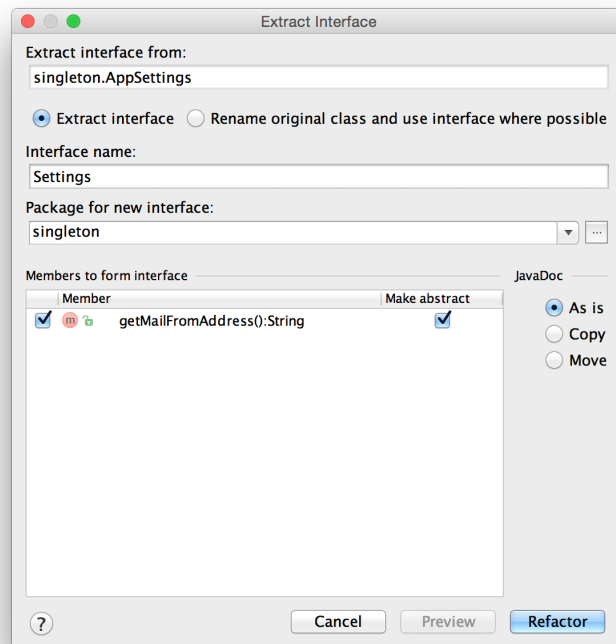
An IntelliJ refactoring exists to do this (see figure 5.7). The first step in this refactoring is to extract a new interface (or convert the old class to an interface) (see figure 5.7a). In the second step, IntelliJ finds all the places in the code where the new interface could be used instead of the concrete class (see figure 5.7b).

In our `AppSettings` and `Mailer` example, this process therefore consists of the extraction of a new `Settings` interface, followed by the usage of the new interface wherever it is possible, including in the constructor parameter of the class being refactored.

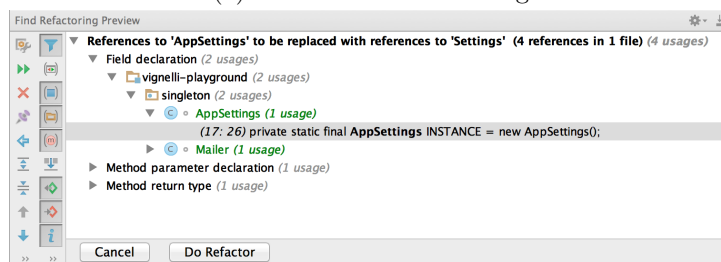
Similarly to type migration to an existing interface, the resulting code can be seen in figure 5.17.

Refactoring Step Goal Checker

Since the goal of this step is the same as the simple “Type Migration” refactoring step, *Vignelli* uses the same goal checker for both refactoring steps.



(a) Extract interface dialog



(b) Use new interface wherever possible

Figure 5.7: Extract interface and use wherever possible IntelliJ refactoring

Chapter 6

Long Method Detection

One of the objectives of this project is to “provide extension points to support identifying other design problems in the future”. To achieve this, it was important to design the system in a way that allows developers to use different techniques to identify different problems.

So far, the techniques we have used to identify train wrecks and direct uses of singletons in *Vignelli* are all based on structure and collaboration templates (see section 2.8.1). Both identification techniques compare a program’s structure to pre-defined problem templates that describe “bad smells”. To highlight the flexibility of *Vignelli*’s architecture we also experimented with the use of a metrics-based analysis approach to identify the “long method” code smell.

6.1 Implementing Existing Metrics-Based “Long Method” Detection

Recall from section 2.9.2 the probabilistic model for deciding whether a method should be considered “long”. [15]

$$IsLongMethod = f(z) = \frac{1}{1 + e^{-z}}$$
$$z = -11.336 + -0.057 * LOC + 4.701 * NBD + 0.598 * VG + 0.486 * PAR$$

Although this model is based on the analysis of only a single project using only three experts to classify methods, we decided that it would nevertheless be interesting to see the results of applying this model to other projects.

Recall from section 3.2.2 that *Vignelli*’s identification engines are able to perform analysis on the PSI tree of a method in order to identify potential problems. Each `IdentificationEngine` is free to analyse the code being passed to its `process()` method in whatever way possible. It was therefore easy to implement the metrics-based approach to identify “long methods” described by Bryton, Brito E Abreu, and Monteiro. To do this, we implemented several utility functions to gather the relevant metrics. Since IntelliJ’s API already provides functionality to generate control flow graphs, even the calculation of the cyclomatic complexity was straightforward to implement.

In *Vignelli*’s user interface we have decided to refer to “long methods” as “complex methods”. This is the result of user testing as some users had preconceptions about what the word “long”. Figure 6.1 shows a positive identification of a complex method that is highlighted by *Vignelli* using the techniques described here.

```

private void processMakerNote(int subdirOffset)
{
    // D. Complex method more... (%F1) and makernote format
    Directory exifDirectory = _metadata.getDirectory(ExifDirectory.class);
    if (exifDirectory==null) {
        return;
    }

    String cameraModel = exifDirectory.getString(ExifDirectory.TAG_MAKE);
    if ("OLYMP".equals(new String(_data, subdirOffset, 5))) {
        // Olympus Makernote
        processDirectory(_metadata.getDirectory(OlympusMakernoteDirectory.class), subdirOffset + 8);
    } else if (cameraModel!=null && cameraModel.trim().toUpperCase().startsWith("NIKON")) {
        if ("Nikon".equals(new String(_data, subdirOffset, 5))) {

```

Figure 6.1: *Vignelli's* long method detection has identified a particularly long method

6.2 Exploring the Validity of the Probabilistic Model

Since *isLongMethod* calculates a probability, it is necessary to define a threshold above which a given method should be considered “long”. In their paper “Reducing subjectivity in code smells detection: Experimenting with the Long Method,” the authors use the cutoff value 0.5 — methods that score higher than 50% with *isLongMethod* are considered “long”. Since the “long method” detection is a binary classification problem, this cutoff point is sensible.

A full statistical analysis of how well the calibrated model by Bryton, Abreu and Monteiro performs on other projects is outside the scope of this project. We have nevertheless analysed a number of methods to explore the validity of suggested model in a practical application.

The method in listing 6.1 counts the number of elements in a given array that are larger than some given threshold. This implementation features a very common code pattern in Java: iterating over a collection and checking a condition for every element.

```

1 private int countLarge(int [] arr, int threshold) {
2     int largeCount = 0;
3     for (int i = 0; i < arr.length; i++) {
4         if (arr[i] > threshold) {
5             largeCount++;
6         }
7     }
8     return largeCount;
9 }

```

Listing 6.1: Method to count elements in array that are larger than some threshold

Applying the *isLongMethod* function to this method results in a calculated probability of 71.44% that `countLarge()` should be considered “long”.

Listing 6.2 shows the same code again, only this time using references to instance variables for `arr` and `threshold`. Since *isLongMethod* also depends on the number of parameters of the method (PAR), the resulting probability is different. In fact, this time, we found it to be 48.63%. This differs significantly from the very similar method in listing 6.1.

It is interesting to find such disparate probabilities for methods that look very similar to the human eye. Given that both `countLarge()` implementations feature the very common “iterate and check” Java code pattern, we believe neither method should be highlighted by *Vignelli* as “long”.

Tests of the model using the example code that was discussed in section 2.4.3 (reproduced here in listing 6.3) also generated surprising results. According to *isLongMethod*, the proba-

```

1 private int countLarge() {
2     int largeCount = 0;
3     for (int i = 0; i < arr.length; i++) {
4         if (arr[i] > threshold) {
5             largeCount++;
6         }
7     }
8     return largeCount;
9 }

```

Listing 6.2: Count large elements in array declared as field on the containing class

bility that the `rtrim()` method should be considered “long” is only 3.55%. This is in contrast to what we presented in section 2.4.3.

```

1 protected String rtrim(String s) {
2     // if the string is empty, do nothing and return it
3     if ((s == null) || (s.length() == 0)) {
4         return s;
5     }
6
7     // get the position of the last character in the string
8     int pos = s.length();
9     while((pos > 0) && Character.isWhitespace(s.charAt(pos - 1)) {
10         --pos;
11     }
12
13     // remove everything after the last character
14     return s.substring(0, pos);
15 }

```

Listing 6.3: Long method example

6.3 Towards a Generalised Probabilistic Model

As noted above, the probabilistic model described by Bryton, Brito E Abreu, and Monteiro is only calibrated based on one particular project by three experts. We therefore do not expect it to be representative of all projects. Indeed, albeit lack of statistical significance, the results of the spot tests we performed above suggest that the model may have problems with common patterns such as “iterate and check”.

Pessoa, Abreu, Monteiro, *et al.* have expanded on [15] in their 2012 paper “An Eclipse Plugin to Support Code Smells Detection,” in which they describe the development of an Eclipse plugin that attempts to create a generalised model using the same technique and continuously improve it using accumulated data from many users. Notably, this work was done under the assumption that such a generalisation exists and can generate good results. Unfortunately, we have been unable to acquire any evaluation data from Pessoa, Abreu, Monteiro, *et al.* that may support their theory.

Because of this, we have added functionality to *Vignelli* to aid in the collection of more metrics data and expert classifications. In the future, we plan to use this data to evaluate whether:

- There is a statistically significant difference between probabilistic models that are generated from data from different projects
- The calibrated models depend significantly on the expert classifying the examples

- A generalised model, that gives good results in practice, exists

In the following section, we outline *Vignelli*'s data collection functionalities.

6.3.1 Method Metrics Collection Using *Vignelli*

We have implemented an IntelliJ action that collects a set of metrics for every method in the currently-opened project. The tool collects the following metrics:

- Lines of code (LOC)
- Lines of comments
- Cyclomatic complexity (VG)
- Number of parameters (PAR)
- Nested block depth (NBD)

Notice the addition of ‘number of lines of comments’. We have decided to collect this metric as comments play a central role in Fowler and Beck’s original description of a long method. Incorporating this metric in the model may lead to improved results.

The “Collect Code Metrics” action can be launched via the IntelliJ **Analyse** menu under the *Vignelli* submenu (as seen in figure 6.2).

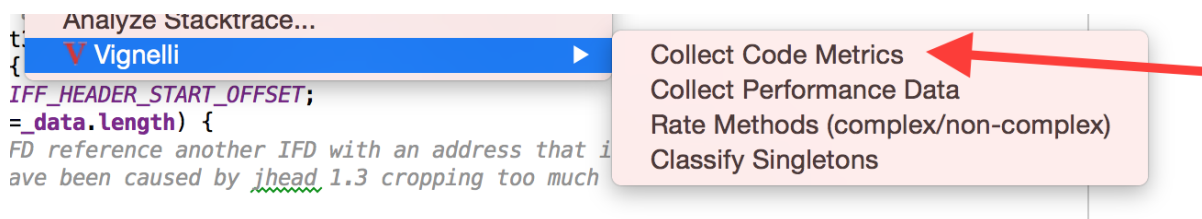


Figure 6.2: *Vignelli* “Collect Method Metrics” action

After the aforementioned metrics have been collected, *Vignelli* writes the results to a JSON file that can be analysed later. An example of the JSON file can be found in listing 6.4.

6.3.2 “Long Method” Classification Collection

We have also implemented another IntelliJ action to gather expert classification data for every method in a project. Once launched, this action visits every method in the project and asks the user to classify the method as either “long” or not (see figure 6.3).

The results of these classifications is again written to a JSON file for later analysis.

6.3.3 Analysis Scripts

Further, we have prepared a range of scripts written in *Ruby*¹ and *R*². These scripts simplify the combination of the collected metrics and classification data, and automatically calibrate new BLR models using the data.

¹<https://www.ruby-lang.org/en/>

²<http://www.r-project.org>

Listing 6.4: JSON representation of method metrics

```
1 {
2   "name": "project-name",
3   "classMetrics": {
4     "ClassNameA": {
5       "name": "ClassNameA",
6       "methodMetrics": {
7         "reverse": {
8           "linesOfCode": 7,
9           "cyclomaticComplexity": 3,
10          "numParameters": 1,
11          "nestedBlockDepth": 2,
12          "linesOfComments": 0
13        },
14        ...
15      }
16    },
17    ...
18  }
19 }
```

6.3.4 Preliminary Results and Outlook

We have already gathered metrics and classification data from three different projects (JPhotoalbum, JUnit, jetty-server). So far, the results are inconclusive; further data collection and statistical analysis is required before we are able to determine in what way the results depend on the project and expert. We have therefore also not yet been able to identify a generalised model that delivers accurate classifications exists.

Should a generalised model be found, we plan to include it in future releases of *Vignelli*. If, on the other hand, no common coefficients between projects and developers can be determined, we still believe that automatic “long method” detection is worthwhile on a per-project basis.

This could be realised by including ways for *Vignelli* users to modify the coefficients manually. For example, a team may stress the importance of having “low nested block depths” for their methods and drag a slider to increase the contribution of this metric to the analysis.

Refactoring Suggestions

Of course, the identification of potentially “long methods” is only one half of what *Vignelli* is designed to do. Once a long method has been identified, multiple techniques can be used to suggest ways to extract parts of methods in order to reduce complexity.

One such technique is used in *JDeodorant* (see section 2.7.5) and is based on the extraction of full computations of variables. Details of this approach are outside the scope of this project but can be found in [40].

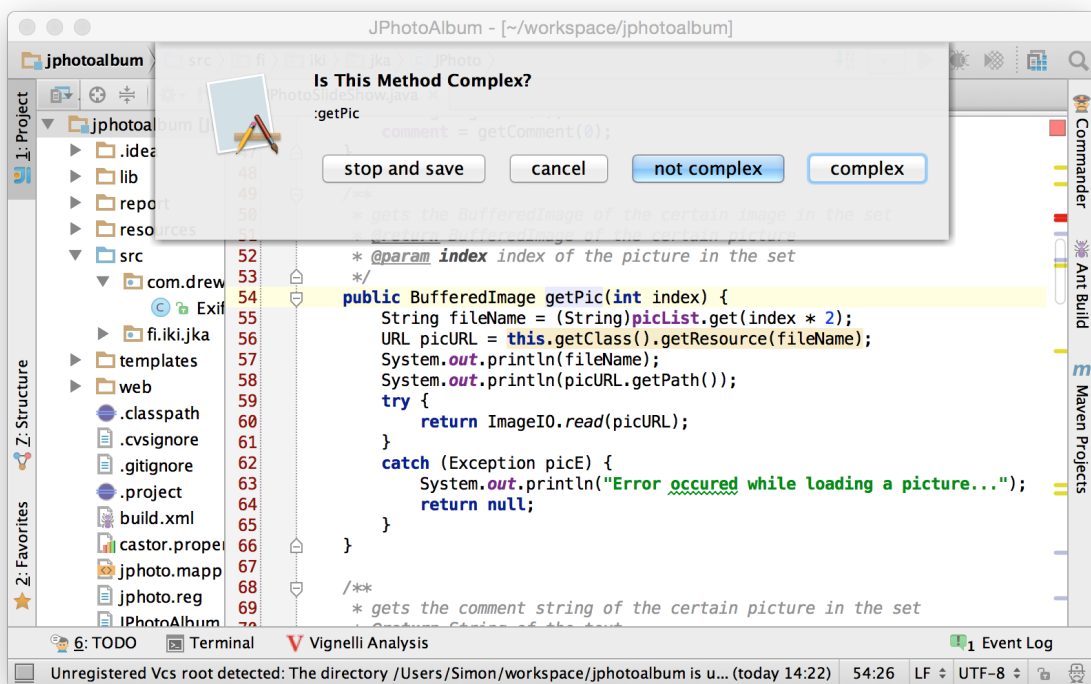


Figure 6.3: Classification of methods through user input

Chapter 7

Evaluation

We have used a variety of evaluation techniques to assess the following aspects of *Vignelli*:

- Train wreck identification and refactoring capabilities (see section 7.1)
- Direct use of singleton identification and refactoring capabilities (see section 7.2)
- Usability of the plugin in terms of clarity to users (see user study, section 7.3) and performance (see section 7.4)

All quantitative evaluations were performed using the same test set up (see appendix A for details).

7.1 Train Wrecks

7.1.1 Identification of Train Wrecks

Evaluation Approach

To evaluate how well *Vignelli* is able to identify train wrecks, we used *Vignelli* to analyse a number of method call chains that appear in the open source *jetty-server*¹ project. *Jetty* is an HTTP server that is under active development and features a large enough codebase that we were confident to find a large variety of samples to analyse that may uncover many edge cases.

For the analysis we proceeded in the following way:

In the first step of our evaluation process, we built an IntelliJ action that automatically classifies all of those method call chains in the project that are train wreck candidates. The only condition a method call chain has to fulfil in order to be considered by this IntelliJ action is that the number of method calls and variable qualifiers must be 3. For example, `customer.getAddress().getZipCode()` matches this description.

Overall, *jetty-server* contained 826 method chains that were train wreck candidates. Since the evaluation of the the identification requires manual classification and the detailed analysis of all method chains was outside the scope of this project, we generated a random sample of 100 of the train wreck candidates. All further analysis was then performed on this data set of 100 train wreck candidates.

¹<http://www.eclipse.org/jetty/>; last accessed: 14 June 2015

Each of the method chains in the data set was automatically classified using the *Vignelli* `TrainWreckIdentificationEngine`. We then also classified each of the method chains manually in order to be able to compare the results with *Vignelli*'s classification.

Evaluation Results

Table 7.1 shows the results of our train wreck identification evaluation.

	<i>Vignelli</i> Positive	<i>Vignelli</i> Negative
Manual Positive	22	0
Manual Negative	0	78

Table 7.1: Train wreck identification evaluation results

As we can see, the results indicate a perfect accuracy in the classification of train wrecks. Recall from section 4.1 that a perfect accuracy of 100% cannot be achieved. This indicates that our test sample is too small to show some of the edge cases that *Vignelli* is unable to detect.

One of these edge cases involves the use of the builder pattern. Consider the code shown in listing 7.1, which uses the builder pattern to construct a `Request` instance, only to call the `send()` method on the newly-constructed object. Recall from section 4.1.5 that the `send()` method of type `void` tells the new `Request` instance to `send()` itself. As discussed in that section, it semantically does not make sense to tell the builder to `send` instead. In other words, we should continue to ask the builder to construct the `Request` instance.

```

1 Request.Builder builder = new Request.Builder();
2 builder.withUrl("http://example.com");
3 builder.withContent("");
4 ...
5 builder.build().send();

```

Listing 7.1: Builder pattern used over multiple lines leads to false positive train wreck identification

In this example, the builder call chain has been split over multiple lines, i.e. the `Request` properties are added in multiple statements instead of one method call chain. The `Request` instance is then constructed on line 5 and told to `send()`. Unfortunately, *Vignelli* cannot treat this method chain (`builder.build().send()`) as a special case. The plugin is unable to distinguish between this and any other short train wreck of length 3, because the calculated type difference is 2. This value is the required type difference to cause a method chain of length 3 to be positively identified as a train wreck (see section 7.1), making this case an example of a false positive identification.

Similarly, despite not showing up in the test data set, *Vignelli* is also prone to false negatives. To illustrate this, consider the code in listing 7.2. This example code constructs a linked list by linking three `ListNode` instances via their `next` fields before finally traversing three nodes to retrieve the value that is stored in the third node.

Figure 7.1 shows the actual object navigation structure of the method call chain that traverses the linked list (highlighted in pink). Clearly, this call chain should be highlighted as a train wreck, as the `LinkedListExample` communicates with all other objects. However, *Vignelli* approximates the object navigation structure by calculating the type difference (see definition 2) of the chain. Since the `getNext()` method is defined on the `ListNode` class and returns an object of the same type, the type difference of the method call chain is 1, which is below


```

1 class LinkedListExample {
2     public void traverse() {
3         ListNode first = new ListNode("First");
4         ListNode second = new ListNode("Second");
5         ListNode third = new ListNode("Third");
6         first.next = second;
7         second.next = third;
8
9         first.getNext().getNext().getValue();
10    }
11 }
12 class ListNode {
13     ListNode next;
14     String value;
15
16     public ListNode(String value) {
17         this.value = value;
18     }
19
20     public ListNode getNext() {
21         return next;
22     }
23
24     public String getValue() {
25         return value;
26     }
27 }

```

Listing 7.2: Example of a train wreck that *Vignelli* does not detect

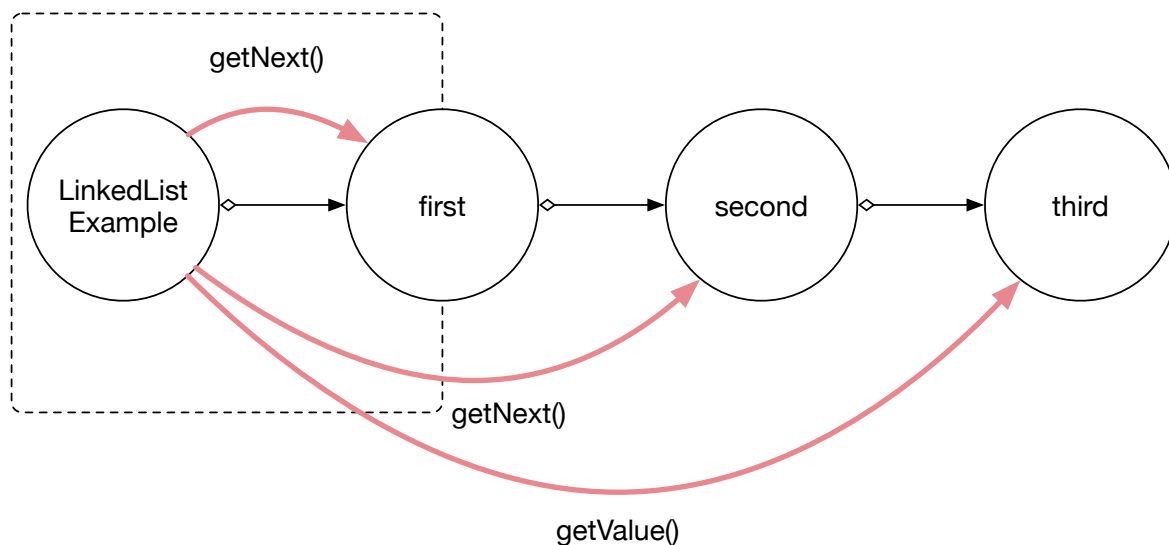


Figure 7.1: Object navigation structure of linked list traversal

the threshold for the chain to be classified as a train wreck. Manual traversal of linked lists is therefore an example of a source for false negatives in *Vignelli's* train wreck detection.

Based on these examples for false positives and false negatives, we can be certain that the true classification rate on a sufficiently large sample (e.g. several thousand method call chains) will be below 100%. However, the results clearly show that these counterexamples are not common in code. This shows that *Vignelli's* train wreck identification algorithm is sufficiently accurate to be used in practice.

Project-External Calls

Interestingly, *Vignelli* only identified 22% of all method chain candidates in our example data set as train wrecks. Random spot checks further indicated that the large number of true negatives could also not entirely be attributed to method chains involving fluent interfaces (such as the builder pattern). Since we had initially expected *Vignelli* to make more positive identifications, we analysed the sample data further.

As it turns out, 11 out of the 78 method chains that were identified as not being train wrecks featured references to external project dependencies. This only includes references to external libraries that are not defined in any `java.*` package.

For example, consider the code in listing 7.3, taken from *jetty-server*'s `InputStreamWritingCB` class.

```
1 _channel.getByteBufferPool().acquire(getBufferSize(), false);
```

Listing 7.3: Train wreck involving external project dependencies

In this method call chain, `_channel` is of type `HttpChannel`, `getByteBufferPool()` returns a `ByteBufferPool` instance, and `acquire()` returns a `ByteBuffer` instance. Out of these three types, only `HttpChannel` is defined in *jetty-server*. On the other hand, `ByteBufferPool` is declared in the *jetty-io* module, which is added as a dependency. The `ByteBuffer` type is defined in Java's own `java.nio` package.

As discussed in section 4.1.7, *Vignelli* does not show train wrecks involving external libraries. Since *jetty-io* is added as a dependency, the method call chain in listing 7.3 is not identified as a train wreck.

Having analysed all 11 of the method call chains that are identified as no train wrecks because of external library dependencies, we noticed that most of these dependencies are contained in modules owned by the *Jetty* organisation. One may therefore argue that these train wrecks should actually be highlighted in *jetty-server* as the *Jetty* development team are able to modify all modules.

In future releases, we plan to include more fine-grained settings that will allow exceptions to be defined for the suppression of train wrecks involving types that are defined in external libraries. These exceptions could be defined on a per-dependency basis.

7.1.2 Refactoring of Identified Train Wrecks

Evaluation Approach

To evaluate *Vignelli*'s refactoring capabilities we used the data set of the 22 positively identified train wrecks from the previous evaluation of *Vignelli*'s identification capabilities. For each of the positively-identified train wrecks we manually launched *Vignelli*'s suggested refactoring process and attempted to step through the steps.

Evaluation Results

Table 7.2 shows the results of attempting to step through *Vignelli*'s suggested refactoring steps for each of the 22 identified problems.

As we can see, we were able to complete 11 (50%) of the 22 refactorings successfully.

Successful Refactorings	Failed Refactorings
11	11

Table 7.2: Train wreck refactoring results

Since we had hoped for a better success rate we investigated the root causes of the failed refactorings further. Unfortunately, we determined a wide variety of small edge cases that caused many of the failures, which we will explain here.

Introduction of Parameter for void Method Call

As explained in section 4.2.5, *Vignelli* attempts to introduce parameters for all fields and member methods that are called in the extracted method so that it can later be moved. However, this technique does not take void methods into account. Once called, these methods do not return a value that can be introduced as a parameter. 2 of the 11 failures can be explained with this edge case.

Covering this case is not straightforward as void methods typically modify the current object's state. This indicates that the coupling between the current class and the class that is being asked for data is more severe. One method that may *sometimes* generate good results is to move the void method as well as the object state that it modifies to the new class as well. However, this is by no means a general solution as this state may still be required by other parts of the current class.

Introduction of Parameter for Member Method Call With Local Arguments

Another problem related to the introduction of parameters was responsible for one failed refactoring. To illustrate this problem, consider the newly-extracted `xyz` method (due to be moved to another class) in listing 7.4. Here, the method calls out to another member method `computeSquare()`. Unfortunately, this method is called multiple times with arguments that are local to `xyz()`. There is no easy way to extract these calls.

```

1 class Example {
2     int computeSquare(int i) {
3         return i * i;
4     }
5
6     void xyz() {
7         for (int i = 0; i < 10; i++) {
8             int square = computeSquare(i);
9             ...
10        }
11    }
12 }

```

Listing 7.4: Member method is called with arguments local to the method to be moved

One potential way to solve this problem is to also move the `computeSquare` method to the new class. However, this is not a general solution, as, again, `computeSquare` may modify the state of the `Example` object.

Moving Method onto Interface Type

A third problem that we encountered was that *Vignelli* attempted to move a method onto an interface type. Consider the modified `ZipCodeExample` implementation in listing 7.5. Here, the `Customer` is an instance of a `Person`. Moving the `xyz` method onto the `Person` interface is not supported by *Vignelli*. Future versions may implement support for this case by moving the method onto all implementations of `Person`.

```
1 class ZipCodeExample {
2     void prepare() {
3         Person customer = new Customer();
4         Label label = new Label();
5         xyz(label, person);
6     }
7
8     void xyz(Label label, Person person) {
9         label.addLine(person.getAddress().getZipCode().toString());
10    }
11 }
12
13 class Customer implements Person { ... }
14
15 interface Person {
16     Address getAddress();
17 }
```

Listing 7.5: Method is about to be moved onto an interface type

Train Wrecks Assigned to Fields

While *Vignelli* is able to refactor train wrecks being assigned to local variables, we are unable to refactor code that involves train wrecks being assigned to fields. Fields represent state of an object and therefore usually cannot be inlined. This means that *Vignelli* is unable to focus the effects of the train wreck on only a small block of code, as it can when inlining a local variable. Again, there is no easy general solution to this problem as the coupling between the current class and those involved in the train wreck is rooted deeply in the structure of the current class.

Other Reasons for Failure

We have found a number of other reasons that led to the failure of the refactoring. These failures ranged from not being able to extract methods, to train wrecks occurring as arguments to calls to other constructors (this happens when constructors delegate some of the initialisation to other constructors via a `this(...)` call).

Closing Remarks

As we can see, there are many edge cases that can lead to the failure of a refactoring, even on such a small sample size. Covering all edge cases is desirable for stability, but also very difficult. We therefore believe that a success rate of 50% is a good achievement, considering the scope of this project. However, it is clear that future work is required to ensure that the refactoring assistance is reliable enough to be truly helpful to software engineers.

7.2 Direct Use of Singleton

Recall from section 2.3.1 that many different implementations of the singleton pattern exist in practice. However, having analysed four open source projects², we have found that the implementation of the singleton pattern is typically consistent within a project, i.e. only one of the approaches described in section 2.3.1 is used in any one project.

As discussed in section 5.1, *Vignelli* is only designed to identify the Java translation of the standard UML description of the singleton pattern. In this evaluation, we will therefore only evaluate the identification of direct uses of singletons that are implemented using this pattern.

7.2.1 Identification of Direct Uses of Singletons

Evaluation Approach

Jetty's implementation consists of a large number of modules, some of which contain classes that implement the singleton pattern. Since *Jetty*'s singletons follow the standard implementation pattern that *Vignelli* is designed to identify we have analysed four modules of the project that contain and use singletons:

- jetty-server
- jetty-monitor
- jetty-osgi
- jetty-start

We have collected evaluation data from these modules in the following way:

- Every static method call (as it is potentially an instance retrieval call) in each analysed module was recorded,
- *Vignelli*'s singleton call identification engine was used to classify each of the static calls,
- We manually classified each of the static calls.

In order to help us perform this classification, we have added an additional IntelliJ action to the plugin. When run, this action navigates to every static call, asks the user to classify it and also performs the automatic classification (see screenshot in figure 7.2).

The results of this classification are written to a JSON file for later analysis. We have written a *Ruby* script to analyse the data.

²Jetty (<http://www.eclipse.org/jetty/>), JDeodorant (<http://jdeodorant.com>), IntelliJ IDEA (<https://www.jetbrains.com/idea/>), Wildfly (<http://wildfly.org>)

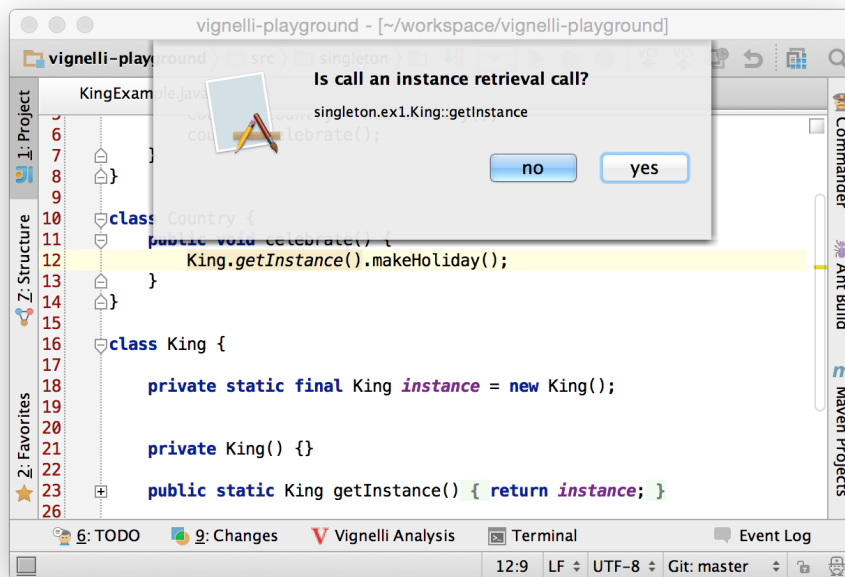


Figure 7.2: Manual classification of instance retrieval call

Evaluation Results

Overall, we analysed 5,213 static method calls in the four *Jetty* modules. Table 7.3 shows the results of the singleton identification evaluation.

	<i>Vignelli</i> Positive	<i>Vignelli</i> Negative
Manual Positive	38	8
Manual Negative	3	5164

Table 7.3: Direct use of singleton identification evaluation results

Based on the “Manual Negative” classifications we can see that the vast majority of all static calls (5167 out of 5213, 99.12%) do not actually retrieve singleton instances. Although this number may be interpreted to mean that the “direct use of a singleton” is not a problem that occurs often in practice, we still believe that it is a problem worth solving, as its ramifications are significant, as explained in section 2.4.2.

Based on these results we are able to get the following measures for the performance of the identification of direct uses of singletons:

Accuracy (ACC)	0.9979
Precision (PPV)	0.9268
Sensitivity (TPR)	0.8261
F1	0.8736

Although we have listed accuracy in this table, we do not believe that this measure is useful to evaluate the performance of the singleton use identification, because of the large number of negative examples. Since *Vignelli* uses structural analysis, these negative examples are very unlikely to be falsely identified as positive examples (only 3 examples). The large number of negative examples can therefore skew the accuracy.

Instead, we think that precision, sensitivity and therefore also F1 are better measures for the performance of the identification technique as these do not take true negatives into account.

Given that the identification of direct uses of singletons uses only very simple techniques, we believe that a precision of 0.9268 and a sensitivity of 0.8261 are good results.

Investigations into the cause for the 8 false negatives (“*Vignelli* Negative”, “Manual Positive”) resulted in the following observation: Some classes that implement the singleton pattern do not feature a `private` constructor. For example, *jetty-server*’s `JMXMonitor` class contains a `public` constructor instead which is annotated with the following comment:

```
/**
 * Constructs a JMXMonitor instance. Used for XML Configuration.
 *
 * !! DO NOT INSTANTIATE EXPLICITLY !!
 */
```

`JMXMonitor` exposes its constructor so that it can be used for XML configuration. *Vignelli*’s detection therefore could be changed to be less stringent about the visibility of the constructor in the future, should this be a prominent problem in many projects.

The 3 false positives (“*Vignelli* Positive”, “Manual Negative”) were introduced to the data through `static` helper methods on the singleton classes themselves. For example, *jetty-server*’s `ShutdownMonitor` features a `static isRegistered` helper method (see listing 7.6). This method uses the `getInstance()` method (highlighted in pink). Although this call technically retrieves the instance of the singleton, it does not introduce any coupling between classes — the method is *contained* within the singleton class itself.

```
1 public static synchronized boolean isRegistered(LifeCycle lifeCycle)
2 {
3     return getInstance() . _lifeCycles . contains (lifeCycle);
4 }
```

Listing 7.6: Static helper method uses `getInstance()`

It is easy to extend *Vignelli*’s identification logic for the direct uses of singletons to exclude those inside the singleton’s themselves. We therefore plan to include this feature in future versions of *Vignelli*.

7.2.2 Refactoring of Identified Direct Singleton Uses

Evaluation Approach

Having identified 38 direct uses of a singleton correctly we then used these calls to evaluate *Vignelli*’s refactoring capabilities.

As discussed in section 5.2.7 some of the refactoring step goal checkers are implemented to search for any remaining references to the old singleton class when an interface should be used instead of the singleton class. This means that here may be multiple singleton retrieval calls in one class that may interfere with one another in this evaluation if one of the calls cannot be refactored successfully. For this reason, before every refactoring test, we have commented out all other singleton retrieval calls in the same class when there were any. Proceeding in this way meant that we could test each of the refactorings in isolation.

Evaluation Results

Table 7.4 shows the results of attempting to step through *Vignelli*'s suggested refactoring steps for each of the 38 identified problems.

Successful Refactorings	Failed Refactorings
28	10

Table 7.4: Direct use of singleton refactoring results

As we can see we can successfully step through 28 of the 38 suggested refactorings (73.68%). Having analysed the 10 refactoring processes that could not be completed we identified the following edge cases that *Vignelli* is unable to cover at this stage.

Multiple Constructors Delegating To Each Other

In section 5.2.3, we discussed that the refactoring process' goal checker waits for *all* constructors to contain the initialisation of the new field for the singleton.

Code listing 7.7 shows how one constructor delegates some of the initialisation to another.

```
1 class KingExample {
2     public KingExample() {
3         this("Default");
4     }
5
6     public KingExample(String name) { ... }
7 }
```

Listing 7.7: Constructor delegation means field will not be initialised explicitly in *all* constructors

Following *Vignelli*'s suggestions, the developer will likely launch IntelliJ's refactoring dialog to convert the instance retrieval method into a constructor-initialised field. However, this will only add the initialisation in the more general `KingExample(String name)` constructor — IntelliJ is intelligent enough to only add the field assignment where it is absolutely required. On the other hand, *Vignelli* lacks the capabilities to perform the control flow analysis that would be required to identify this case.

Since the goal checker condition is not satisfied, *Vignelli* will remain stuck on the first refactoring step, although the manual addition of the initialisation to *every* constructor will progress the refactoring process to the next step.

Calling Protected Methods on Singleton in the Same Package

Consider the code in listing 7.8 that defines the `AppSettings` singleton, only this time declaring `getMailFromAddress()` protected.

Making this method protected results in IntelliJ not being able to include its definition in an interface that it can extract — only public methods can be included. To extract a sufficient interface, *Vignelli* would therefore also have to escalate the visibility to `public`. This is currently unsupported. However, it is worth noting that this limitation was the cause for 9 of the failed refactorings. Adding this edge case should therefore be a high priority in future versions of the software.


```

1 class AppSettings {
2     private static final AppSettings INSTANCE = new AppSettings();
3
4     private AppSettings() { }
5
6     public static AppSettings getInstance() { ... }
7
8     protected String getMailFromAddress() { ... }
9 }

```

Listing 7.8: protected singleton method

7.2.2.1 Goal Checker Check All Limitation

As indicated in the introduction to this section, the type migration goal checker is extremely imprecise. This is because it only considers a type migration to be successful when *no more* references to the previous class exist in the current class. In large classes, this is a big limitation.

However, it is important to note that in order to improve the goal checker significantly, we would be required to perform a more rigorous control flow analysis of which type references should actually be migrated. This task is outside the scope of this project.

7.3 User Testing

We have relied heavily on a user-centred design process to ensure that *Vignelli* can be used easily by developers of all experience levels. To do this, we have used the following three approaches:

- We have used *Vignelli* ourselves in our own development,
- One long-term tester has run the plugin in their development setup to report potential platform-dependent bugs and also report their views on the UI design of the software,
- We held a user study to gather feedback from a variety of potential users of our software and to evaluate their first-time usage behaviour.

7.3.1 Iterative Improvements Through Continuous Feedback

Since we and an external tester started using *Vignelli* from very early on in its development, we have been able to identify and improve on a number of usability concerns. These included:

- Previous versions of the plugin included a way for *Vignelli* to assist in multiple refactorings at the same time. Although this design choice was made to enable developers to perform short refactorings in the middle of other refactorings, it very quickly became apparent that this feature was not useful. Instead, it increased the complexity of the UI, distracted from one refactoring, and also increased the number of crashes significantly.
- Previous versions of the plugin did not feature refactoring step explanations that are tailored specifically to the code being refactored. Generic explanations made it difficult to follow instructions. For example, explanations such as “extract an interface from the singleton class and use it wherever possible” did not explicitly name the right classes which made following the refactoring steps more difficult. We solved this by including domain-specific information in the refactoring step explanations, such as the names of classes or

even full methods (see figure 7.3).

Next Step: Extract Interface for Singleton and Use it Wherever Possible

In this final step we now decouple `Country` from `King` altogether by extracting an interface from it and using that whenever possible.

`King` in this case will simply be just another instance of the interface that we extract, i.e. `King` will implement the newly extracted interface.

Figure 7.3: Explanations feature references to existing classes

7.3.2 User Study

As well as using our own feedback and that of our external tester to continuously iterate on *Vignelli*'s user interface, we also took part in the departmental "Project Fair" where we gathered a wider range of feedback from users of different levels of experience.

Overall, we interviewed 11 students, 10 of whom were from year 1–4 as well as one MSc student. Although we had originally planned to ask more students for feedback, we found that, we frequently received many of the same comments. With the test subject's consent, we recorded their interactions as a screencast (audio and video). This allowed us to focus on the user and watch them more closely as well as review the material later, instead of taking notes during the study.

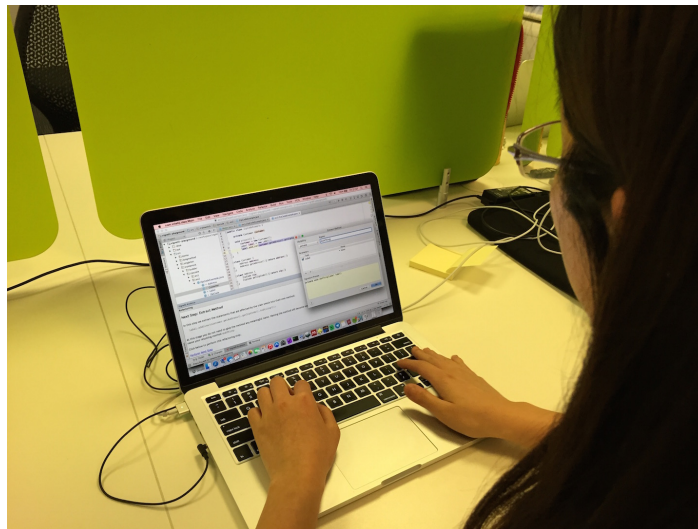


Figure 7.4: Test Subject Using *Vignelli* to Refactor Sample Code

Each user study consisted of three sections:

1. Background Questions,
2. Interaction with the *Vignelli* IntelliJ plugin,
3. Follow-up Questions about Users' Impressions.

Background Questions

The original premise for developing *Vignelli* was that early feedback on design decisions will help software engineers learn about design. To validate this premise we first asked each user what they considered their biggest problem regarding code design.

Two test subjects answered that they tended to over-engineer their software and spend a long time attempting to figure out an extendible design up front. One of the respondents noted that they did not know when to start designing and that they often ended up with very few classes with very large methods. All other respondents said that they felt as though they often missed the point at which they *should have* refactored code. One of the respondents noted that they were unsure where to start refactoring to improve the design and therefore did not. According to them, this often resulted in a “big ball of mud”.

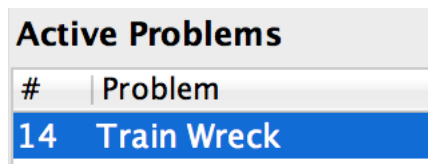
Although a set of 11 students is by no means a representative sample of students, these responses are a good indication that *Vignelli*'s early feedback on design problems would be able to help at least 8 out of the 11 test subjects.

We then presented test subjects with a piece of example code that contained a train wreck (see listing 4.1) and asked them to identify any potential design problems in the piece of code. Out of the 11 respondents, only one of the respondents was able to identify the train wreck and provide an alternative implementation without train wrecks. When prompted, 6 out of the 11 respondents had not heard of the “train wreck” code smell before.

Interaction with *Vignelli*

To evaluate the quality of *Vignelli*'s problem identification user interface, we asked users to identify train wrecks as well as direct uses of singletons in example code.

- All test subjects found the plugin's use of IntelliJ inspections intuitive and were able to find all problems.
- One user commented that the use of “#” in the line number column of the *Vignelli* tool window was confusing — this particular user thought the number corresponded to the number of instances of that particular problem in the code.



Active Problems	
#	Problem
14	Train Wreck

- *Vignelli*'s problem list in the tool window changes when the user changes the file they are currently working on. All of the test subjects found this behaviour intuitive.
- Although all of the test subjects found the problem explanation of the “direct use of a singleton” to be informative and understandable, the same cannot be said for the “train wreck” explanation. Only 1 out 11 users could explain the problems associated with a train wreck in their own words after seeing the explanation. Since the user study, we have acted on this feedback and adapted the explanation.

After asking the testers to identify all problems, we instructed them to follow the suggested refactoring steps for both a “train wreck” problem and a “direct use of a singleton”. Since what people say and what they do are often not the same [41], we asked very few questions and intervened as little as possible during this process, instead observing user behaviour.

- Some refactoring steps explanations are too long to fit into the IntelliJ tool window and users are required to scroll through. 3 out of the 11 test subjects did not realise this which caused them to skip parts of the explanations.
- Some of the refactoring step explanations feature lists of actions that the developer is asked to perform. For example, in the “Extract Interface and Use whenever possible” refactoring step, *Vignelli* instructs developers to select certain options in the IntelliJ refactoring dialog. All test subjects responded positively to having such a list of actionable items. However, several users pointed out a number of small inconsistencies between these lists and actual user interface elements in IntelliJ. Since the user study, we have fixed these small issues.
- Two users did not follow the instructions that were given in the refactoring step. For one of the users *Vignelli* cancelled the active refactoring and switched back to observation mode as expected. For the other user, the active refactoring had to be cancelled manually, highlighting the approximate nature of *Vignelli*’s goal checkers. However, no other users ran into issues with the approximate nature of goal checkers.
- All users expressed that they preferred the “longer but clearer“ explanations of the “Direct use of a singleton” refactoring process. We have since enhanced the refactoring step explanations for the “train wreck” refactoring process.
- 5 out of the 11 test subjects noted that after focusing on the *Vignelli* tool window to read the explanation and launch the next refactoring step, it was difficult to identify what exactly had changed in the code. We have received suggestions to include a “before-after” comparison view for each step to see the effects of the last refactoring step.

User Impressions

Overall, the feedback we received for the *Vignelli* plugin was very positive.

- Some students with more programming experience noted that they would primarily use *Vignelli*’s observation mode and are interested in installing the plugin for this reason.
- Some students (mostly the same students with more programming experience) expressed that performing the same refactoring steps multiple times could get tedious and explained that a “quick fix” functionality would be useful. This “quick fix” option would allow *Vignelli* to perform all refactoring steps automatically, thus skipping all explanations. However, we think that this functionality would actually distract from the learning aspect of assisting the user in refactoring their code.
- Many test users explained that *Vignelli* could be very useful for second-year students at Imperial who attend their first software engineering design course during that year.
- 9 out of 11 students expressed that it would be good for *Vignelli* to pre-populate IntelliJ refactoring dialog boxes with the right selections. Unfortunately, as explained in section 3.3.1, this is currently impossible to realise.

7.4 Performance Testing

Since one of our goals was to guarantee continuous feedback on the design of the code that is being written, we have evaluated the performance of all problem identification engines that we have implemented in *Vignelli*.

Experiment Setup

We have implemented benchmarking actions in IntelliJ that help us find statistically significant data on the speed of problem identifications. The performance data was gathered in the following way:

- Three projects were analysed,
- Every method in the project was analysed,
- Every method was analysed by all identification engines separately,
- Every method was analysed 100 times by each identification engine in order to ensure statistically significant results.

To perform the experiments we have built an IntelliJ action³ that, for every method in the currently-opened project, processes that method in all *Vignelli* identification engines and records the execution time of the identification process (this is done 100 times).

We have benchmarked *Vignelli* on the following three open source projects which we have chosen for their differences in code style and popularity.

- JPhotoalbum⁴,
- JUnit⁵,
- jetty-server⁶.

IntelliJ calls the *Vignelli* identification engines' `process` method on a per-method basis. For this reason, we also measure the performance of the identification by processing individual methods. To avoid possible skewing of the data if one project's style meant that its methods contained, on average, more statements than those in other projects, we merged the performance measurements from the three projects above to normalise the data in this way.

7.4.1 Performance Test Results

Overall, we ran all identification engines on 6,236 methods from the three projects, resulting in 623,600 performance measurements. Table 7.5 shows the performance results that we measured running *Vignelli* in the test environment. These results do not take the length and complexity of the analysed methods into account as we would like to get an indication of best-case performance, worst-case performance but also the average case. Analysing methods of different sizes most closely resembles the real-world application of the identification engines. Note that the results in table 7.5 are measured in nanoseconds (ns).

As we can see, the results show first and foremost that there exists a great degree of variance in the performance to identify problems in one method: while some identifications take only very few nanoseconds (as low as 1,000ns) there are some extreme outliers with one train wreck identification taking as long as 1.899s. Although this the maximum time we have measured, we have identified a small number of similar measurements, i.e. this maximum measurement is not a singlet outlier. With regards to our goal, supporting real-time analysis of the code as

³An IntelliJ action can run a specific task when the user launches it, e.g. from the IntelliJ "Analyze" menu.

⁴<http://sourceforge.net/projects/jph/>

⁵<https://github.com/junit-team/junit>

⁶<http://www.eclipse.org/jetty/>

	Train Wreck	Singleton	Complex Method
Min	0.001	0.001	0.007
Median	0.018	0.008	0.062
Mean	0.087540	0.015642	0.244100
Max	1,899	25.533	1,600
Standard Deviation	4.518	0.078	3.596

Table 7.5: Performance results for all methods from all projects combined (times in ms)

it is being written, these large numbers are far too high — real-time applications are generally expected to return results within a few milliseconds [42].

However, note that these outliers are not the norm. In fact, the largest average time it takes any of the identification engines’ to process an average method is only 0.2441ms which we absolutely deem to be acceptable. In fact, though, the median time is even lower, the largest of them being 0.062ms. Clearly, the results vary greatly. More formally, we were able to find a relatively high standard deviation of up to 4.51777ms. Although this is a relatively high number it does not have a significant effect on the user experience — even a few hundred milliseconds delay would still be acceptable.

Note also, though, that in a real-world application, multiple identification engines will be processing methods simultaneously. Since the individual benchmarks are extremely satisfactory and since multiple threads are able to *read* PSI trees simultaneously, we have not attempted to model a real-world scenario in which the identification engines work concurrently. Additionally, during user testing, no test subject expressed concern over the performance of the tool (see section 7.3).

Interestingly, the data also shows that the identification of direct uses of singletons is by far the fastest identification engine. This is despite the fact that for every static method call, *Vignelli* will find the callee and analyse its class structure. In contrast, the train wreck identification can be executed entirely using only the local method call chain. We justify this with the large amount of computation required to identify critical call chains (see section 4.3.1).

Overall, the results show that *Vignelli* is indeed able to give users continuous feedback during their development.

7.5 Stability

For *Vignelli* to be successful as an IntelliJ plugin and be used by developers to learn more about design, the plugin is required to be stable. To evaluate the stability of the plugin we distinguish between observation mode and refactoring mode.

Observation Mode

Since *Vignelli* observes the code the developer is writing most of the time, stability in observation mode is critical. We can report that the analysis of 6,236 different methods resulted in no crashes of the plugin.

Refactoring Mode

Unfortunately, due to the many edge cases that are not covered, *Vignelli* is less stable in refactoring mode. All 11 failed “train wreck” refactorings caused the plugin to crash, as did 5 of the failed singleton refactorings. These results show that many edge cases have to be covered before refactorings can work reliably, an observation that is corroborated by Jemerov in “Implementing refactorings in IntelliJ IDEA” [43].

7.6 Learning Improvement

In section 2.5 we suggested that an accelerated feedback loop with respect to software design, as realised in *Vignelli*, will help developers in their transition to object-oriented thinking.

This aspect of *Vignelli* is one of the most difficult to evaluate as it requires long-term studies of how developers adapt their design thinking based on the feedback they receive from our tool. This could be achieved by observing the frequency at which *Vignelli* identifies problems in the developer’s source code. A decrease in this frequency may indicate a change in the developer’s design thinking.

However, this experiment requires many participants taking part over a long time. Due to these constraints we have not been able to perform such a study.

Chapter 8

Conclusions

We have produced *Vignelli*, an IntelliJ IDEA plugin that helps developers to improve the design of their software by identifying design flaws in code as it is written, informing the developer about these flaws and then assisting them in the refactoring process towards a better design.

8.1 Core Achievements

Fast Software Design Feedback Loop

Vignelli analyses the code that is being written by the developer in real-time and informs him or her of likely design flaws as they appear. This is achieved by performing only a partial analysis of the abstract syntax tree of the program that is being developed; when a method changes, only the source code of that method is analysed for new design flaws.

This continuous nature is what sets *Vignelli* apart from existing tools and processes on the market that aim to improve developers' design sensibilities. By being able to give feedback instantly, we have vastly accelerated the software design feedback loop. By finding out about potential design problems faster than before, developers are able to make quick adjustments to their code and improve their design sensibilities on a continuous basis, as recommended in the literature [1], [10], [18].

Our evaluation shows that we can detect all currently-supported design flaws very efficiently in less than one millisecond on average.

Accurate Identification of Design Flaws

By analysing the structure of method call chains and approximating runtime object relationships, we have been able to identify train wrecks in the source code under development with near-perfect accuracy. In fact, our evaluation data set did not expose any false positives or false negatives, indicating that the approximations that were necessary to be able to statically find train wrecks are good.

Similarly, the detection of direct uses of singletons is based on the analysis of the structure of potential singleton classes. Using a very simple identification technique we have been able to achieve 92.68% precision and 82.61% sensitivity for the identification of one common implementation pattern of the singleton.

Overall, the results show that we have successfully implemented fast, yet accurate techniques to identify design flaws in software projects.

Refactoring Assistance

Once a design flaw has been identified, *Vignelli* assists developers in the refactoring of their code to improve the design. By using composite refactoring techniques in a similar style to those described in *Refactoring to Patterns* [18], *Vignelli* is able to guide the developer through a refactoring process, using standard refactoring techniques.

Each refactoring step can be performed using IntelliJ’s built-in refactoring tools, but also manually. This was achieved by defining goal patterns of what the AST of the code being refactored should look like and then attempting to match these patterns after every code modification. Although we have only been able to approximate the description of these goal patterns, user testing has shown that this very rarely becomes an issue.

Extensible Design

Vignelli is designed in a way that makes it easy to add support for more identifiable problems. Our experimental implementation of a metrics-based analysis approach has shown that *Vignelli* is flexible enough to accommodate different kinds of analysis techniques with ease.

8.2 Future Work

Although we have been able to meet our main objectives to a satisfactory standard, there are a number of ways in which we would like to improve *Vignelli* in the future:

User Interface Improvements

Although we have user-tested our application, there are some user interface improvements that we would like to implement for which we did not have the time:

- We plan to include a “hide problem” feature that allows users to hide individual problems so that they no longer show up. Since any project with a singleton class requires at least one direct instance retrieval call it would be nice for a developer to hide *Vignelli*’s warning in this case.
- We plan to link the code descriptions in *Vignelli*’s tool window back to the original code snippets in the editor. As it is currently difficult to make the connection between the elements, we envision a feature that highlights code in the editor when the developer hovers their mouse over the corresponding code snippet in the tool window.

Support for More Singleton Implementation Patterns

Vignelli is currently able to identify only the standard UML translation implementation of the singleton pattern. Since there are many other ways to implement the singleton pattern we plan to extend *Vignelli*’s “direct use of singleton” identification functionality to also support other singleton structures.

Refactoring: Support Undo Operation

As discussed in section 3.2.7, *Vignelli* currently does not support the *Undo* operation — active refactoring processes can easily get cancelled or even stuck. In the future, we would therefore

like to add support for the undo operation by gracefully reverting to previous refactoring steps.

Refactoring Edge Cases

Even though *Vignelli* is able to assist in the refactoring of many problems, our evaluation has shown that there are many more edge cases that will need to be covered in order for *Vignelli* to be less prone to crashes and hangs during the refactoring process.

Further Analysis of Long Methods

As discussed in section 6, we would like to expand our statistical analysis on occurrences of “long methods” in the attempt to calibrate a generalised binary logistic regression model that is able to detect “long methods” accurately across different projects.

Appendix A

Test Setup

A.1 Hardware Configuration

Model	MacBook Pro (Retina, 13-inch, Mid 2014)
Processor	Intel Core i7 (4578U), 3.0 GHz
Architecture	64 bit
Memory	16GB 1600 MHz DDR3
Storage Type	Flash Storage
Graphics	Intel Iris 5100 (1536MB)

Table A.1: Test hardware configuration

A.2 Software and Environment Configuration

- OS X Yosemite (10.10.3),
- All applications other than IntelliJ (see next item) were closed,
- IntelliJ 14.0.1 process, started from a host IntelliJ 14.0.1 instance, with the *Vignelli* plugin installed,
- All *Vignelli* inspections were turned on,
- For gathering evaluation statistics using IntelliJ actions, no files were open,
- Every test was performed in a fresh instance of IntelliJ to counteract potential memory leaks.

Bibliography

- [1] M. Fowler, *Frequency reduces difficulty*, online, [Accessed 15 June 2015], Jul. 2011. [Online]. Available: <http://martinfowler.com/bliki/FrequencyReducesDifficulty.html>.
- [2] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st. Addison-Wesley Professional, 2010.
- [3] S. McConnell, *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004.
- [4] R. C. Martin, “Design principles and design patterns,” *Object Mentor*, no. c, pp. 1–34, 2000.
- [5] —, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.
- [6] B. Foote and J. Yoder, “Big ball of mud,” *Pattern languages of program design*, 1997. [Online]. Available: <http://files.meetup.com/1286116/ballofmud.pdf>.
- [7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
- [8] J. Bloch, *Effective Java*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.
- [9] Wikipedia, *Singleton pattern — wikipedia, the free encyclopedia*, online, [Accessed 15 June 2015], Jun. 2015. [Online]. Available: https://en.wikipedia.org/wiki/Singleton_pattern.
- [10] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*, ser. Object Technology Series. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
- [11] S. Metz, *Practical Object-Oriented Design in Ruby: An Agile Primer*. Pearson Education, 2012. [Online]. Available: https://books.google.co.uk/books?id=VRCv%5C_bATuSIC.
- [12] K. Lieberherr, I. Holland, and A. Riel, “Object-oriented programming: an objective sense of style,” *ACM SIGPLAN Notices*, vol. 23, no. 1, pp. 323–334, 1988.
- [13] A. Hunt and D. Thomas, *Tell, don't ask*, [Accessed 15 June 2015], 2015. [Online]. Available: <https://pragprog.com/articles/tell-dont-ask>.
- [14] M. Fowler, *Fluent interface*, [Accessed 15 June 2015], 2005. [Online]. Available: <http://martinfowler.com/bliki/FluentInterface.html>.
- [15] S. Bryton, F. Brito E Abreu, and M. Monteiro, “Reducing subjectivity in code smells detection: Experimenting with the Long Method,” *Proceedings - 7th International Conference on the Quality of Information and Communications Technology, QUATIC 2010*, no. 3, pp. 337–342, 2010.

- [16] O. Astrachan, G. Mitchener, G. Berry, and L. Cox, “Design patterns: an essential component of CS curricula,” *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pp. 153–160, 1998.
- [17] A. Chatzigeorgiou, N. Tsantalis, and I. Deligiannis, “An empirical study on students’ ability to comprehend design patterns,” *Computers & Education*, vol. 51, pp. 1007–1016, 2008.
- [18] J. Kerievsky, *Refactoring to Patterns*. Pearson Higher Education, 2004.
- [19] T. Mackinnon, “Endo-testing: unit testing with mock objects,” *Extreme programming . . .*, 2001. [Online]. Available: <http://instinct.googlecode.com/svn/tags/Release-0.1.6/core/docs/reference/endotesting.pdf>.
- [20] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, “Mock roles, not Objects,” *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '04*, p. 236, 2004. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1028664.1028765>.
- [21] E. A. Hunt, D. Thomas, I. T. Pragmatic, A. Hunt, T. Mackinnon, and S. Freeman, “Software Construction,” no. June, pp. 22–24, 2002.
- [22] A. Causevic, D. Sundmark, and S. Punnekkat, “An industrial survey on contemporary aspects of software testing,” *ICST 2010 - 3rd International Conference on Software Testing, Verification and Validation*, pp. 393–401, 2010.
- [23] Wikipedia, *Integrated development environment — wikipedia, the free encyclopedia*, [Accessed 15 June 2015], 2015. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Integrated_development_environment&oldid=643191757.
- [24] J. Mahmood and Y. R. Reddy, “Automated refactorings in Java using IntelliJ IDEA to extract and propagate constants,” *2014 IEEE International Advance Computing Conference (IACC)*, pp. 1406–1414, Feb. 2014. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6779532>.
- [25] Checkstyle, *Checkstyle*, [Accessed 15 June 2015], 2015. [Online]. Available: <http://checkstyle.sourceforge.net>.
- [26] PMD, *Pmd — don't shoot the messenger*, [Accessed 15 June 2015], 2015. [Online]. Available: <http://pmd.sourceforge.net>.
- [27] A. J. Riel, *Object-Oriented Design Heuristics*, 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.
- [28] SourceMaking, *Feature envy*, online, [Accessed 15 June 2015], Jun. 2015. [Online]. Available: <https://sourcemaking.com/refactoring/feature-envy>.
- [29] F. Bergenti and A. Poggi, “Improving UML designs using automatic design pattern detection,” *12th International Conference on Software . . .*, 2000.
- [30] D. Hovemeyer and W. Pugh, “Finding bugs is easy,” *ACM SIGPLAN Notices*, vol. 39, p. 92, 2004.
- [31] N. Tsantalis and A. Chatzigeorgiou, “A Novel Approach to Automated Design Pattern Detection,” *. . . on Informatics (PCI) . . .*, 2005.
- [32] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*, 1st. Springer Publishing Company, Incorporated, 2010.
- [33] T. Pessoa, F. B. E. Abreu, M. P. Monteiro, and S. Bryton, “An Eclipse Plugin to Support Code Smells Detection,” p. 12, 2012. arXiv: 1204.6492. [Online]. Available: <http://arxiv.org/abs/1204.6492>.

- [34] M. Cinnéide, “Towards Automated Design Improvement Through Combinatorial Optimisation,” in *Proc. Workshop on Directions in Software ...*, 2004. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.189.970>.
- [35] Wikipedia, *Abstract syntax tree — wikipedia, the free encyclopedia*, online, [Accessed 15 June 2015], May 2015. [Online]. Available: http://en.wikipedia.org/wiki/Abstract_syntax_tree.
- [36] JetBrains, *Testing intellij idea plugins*, [Accessed 15 June 2015], Jun. 2015. [Online]. Available: <https://confluence.jetbrains.com/display/IDEADEV/Testing+IntelliJ+IDEA+Plugins>.
- [37] Y. Bugayenko, *Oop alternative to utility classes*, online, [Accessed 15 June 2015], May 2014. [Online]. Available: <http://www.yegor256.com/2014/05/05/oop-alternative-to-utility-classes.html>.
- [38] M. Rybak, *Why static code is bad*, online, [Accessed 15 June 2015], Jul. 2013. [Online]. Available: <https://objcsharp.wordpress.com/2013/07/08/why-static-code-is-bad/>.
- [39] K. Nordmann, *Static considered harmful*, online, [Accessed 15 June 2015], Mar. 2011. [Online]. Available: http://kore-nordmann.de/blog/0103_static_considered_harmful.html.
- [40] N. Tsantalis and A. Chatzigeorgiou, “Identification of extract method refactoring opportunities for the decomposition of methods,” *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, Oct. 2011. [Online]. Available: <http://linkinghub.elsevier.com/retrieve/pii/S0164121211001191>.
- [41] J. Nielsen, *First rule of usability? don't listen to users*, online, [Accessed 15 June 2015], Aug. 2001. [Online]. Available: <http://www.nngroup.com/articles/first-rule-of-usability-dont-listen-to-users/>.
- [42] Wikipedia, *Real-time computing*, online, [Accessed 15 June 2015], Jun. 2015. [Online]. Available: https://en.wikipedia.org/wiki/Real-time_computing.
- [43] D. Jemerov, “Implementing refactorings in IntelliJ IDEA,” *Proceedings of the 2nd Workshop on Refactoring Tools - WRT '08*, pp. 1–2, 2008. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1636642.1636655>.