Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Automatic Detection and Correction of Code Smells

Author:
Yuhang Huang

Supervisor:
Dr. Robert Chatley

Second Marker:
Dr. Naranker Dulay

June 18, 2018

Abstract

In this project we develop an IntelliJ plugin that assists developers in the detection and correction of three different code smells in Java code. "Replace conditional with polymorphism", "Excessive cyclomatic complexity" and "Return private mutable field" code smells are detected quickly and accurately. Automatic correction consists of step by step refactoring guidance, which is provided to help developers through the process of removing these code smells.

Analysis of open-source projects shows that these code smells are very common in existing Java code. This suggests that even though many Java developers use powerful IDEs, they do not have tools and techniques to reveal these code smells and to fix them. Our tool addresses this, and we also believe that the guidance provided by our tool may also help developers to improve their ability writing good quality code.

Our tool is able to provide a fast feedback loop to developers, by performing code analysis as the developer is making changes to the code. Most of our code inspections complete within 1 second. The tool provides accurate code smell detection, with over 80% accuracy achieved in the trials we have carried out on popular open-source projects. The more difficult task of automated correction on replace conditional with polymorphism and excessive cyclomatic complexity code smells were also successfully completed in 45% and 84% of cases respectively.

Acknowledgements

I would like to thank my supervisor, Dr Robert Chatley, for his consistent support throughout the project. His comments, suggestions and advice have been invaluable and truly helpful, and I owe a large part of any success of this project to him.

I would also like to take this opportunity to thank my parents and my little sister. They gave me unconditional love and support throughout my four years at Imperial, I would not have made any of these without them.

Contents

Chapter 1

Introduction

1.1 Motivation

Code quality is an essential property of a software, because it has an significant impact on the maintainability, scalability and extensibility of the software. Good quality code is easy to add new features, fix bugs, and modify existing features with a minimum of effort without the risk of breaking other parts of the software. On the other hand, bad quality code is difficult to understand and work with, and potentially leads to financial loses and waste of time for further maintenance and modifications.

In order to maintain good code quality, there are generally two standard approaches widely used in industry. The first one is through manual code review - developers ask another team member to inspect and examine the code they have written. Although this is an effective way to ensure high code quality, this approach is time consuming and labor intensive. The second approach is using code analysis tools. Many tools exist to help developers improving code quality from different aspects. For example, CheckStyle [1] focuses on enforcing coding standards while the C analyzer LINT [2] tends to look for problems with pointer arithmetic, null references etc.

This project focuses on a different aspect of code quality - "code smells", a term first introduced by Martin Fowler in his book [3]. Code smells are warning signs in a piece of code that usually (but not always) indicate a problem. The listing 1.1 below shows an example of using conditional switch statement instead of polymorphism which indicates a poor object-oriented design. Although they are not bugs and they do not currently prevent the software from functioning, they normally indicate weaknesses in design that may be slowing down development or hidden problem that may be increasing the risk of bugs in the future.

Determine code smells accurately is difficult. It requires developers to have practical experience and the experience can only be gained through a long period of time working with real life applications. Developers also need to be familiar with the code base and fully understand relevant design patterns. Developers without a lot of experience might sense something is wrong in a piece of code - the code is smell bad, but struggle to evaluate and reveal the underlying problem or failed to apply refactoring techniques to correct it. In addition, the refactoring of the code smell is almost always case by case, this process is labor intensive and requires significant developer's attention and effort.

This project aims to automate the process of identifying code smells, guide developers to fix code smells through step by step refactoring guidance and eventually help developers to improve on their code quality.

```
public class Bird {
    ...

    public int getSpeed() {
        switch (type) {
            case PENGUIN:
                return 1;
            case EUROPEAN:
                return 2 * baseSpeed;
            case PARROT:
                return baseSpeed > 10 ? baseSpeed : 10;
            default:
                return 0;
        }
    }
}
```

Listing 1.1: Example of using conditional switch statement instead of polymorphism

1.2 Objectives

One of the most important considerations in this project is to benefit a wide range of developers. The tool developed should provide a seamless integration into developer's software development process so that they have an intuitive and streamlined experience using the tool. In addition, the tool should benefit developers regardless the level of their coding experience, the kind of applications they work on, or their development environment operating system.

Another objective is to detect code smells quickly in order to provide a fast feedback loop to the developers during their early stage of software development. The shorter the feedback loop is, the higher chance a code smell is identified early so as to minimize the amount of time and effort spent to remove the code smell.

In addition, code smells should be identified accurately and precisely. False positive and false negative detections should be kept to a minimum so that developers are confident about the use of the tool.

After the code smells are identified, refactoring guidance should be provided to the developers. The tool should also allow developers to learn different refactoring techniques as they are guided through the refactoring process, and apply these techniques to remove the code smell and improve the code quality.

To summarize, the objectives of this project are not only to develop a tool that helps developers to identify code smells, assists them in removing found code smells using different refactoring techniques, but also to reveal the importance of code quality in modern software development, help developers to learn different refactoring techniques through the use of the tool, and improve their ability to write good quality code.

1.3 Contributions

In this project we develop Automatic Code Smell Detector, an Intellij plugin that identifies code smells quickly and accurately, provides refactoring guidance, assists developers to remove code smells, helps developers to learn different refactoring techniques, and improves their ability writing good quality code. The main contributions of this project are below:

- The plugin is able to identify and provide refactoring suggestion to a code smell so called "using conditional instead of polymorphism" which occurs when the behavior of an object depends on one of its own properties and the selection of the behavior is made within a switch statement where polymorphism should be used to adhere to object-oriented design. (see section 4)
- The plugin is able to identify and provide refactoring suggestion to a code smell so called "excessive cyclomatic complexity" which is a metric based analysis used to indicate the number of linearly independent paths within source code in order to reflect its complexity. (see section 5)
- The plugin is also able to identify and provide refactoring suggestion to a code smell so called "return private mutable field" where the reference to a private mutable class member is returned and exposes vulnerability that the class state might be changed unexpectedly. (see section 6)

The plugin is published to JetBrains Plugins Repository (https://plugins.jetbrains.com/) and can be accessed at: https://plugins.jetbrains.com/plugin/10778-code-smell-detector.

Chapter 2

Background

2.1 Code Quality

Software quality is essential in the development of a piece of software, it requires the functional requirements or specifications to be met as well as non-functional requirements such as robustness or maintainability to be achieved[4]. This project aims to improve one aspects of software quality - code quality.

There are as many ways to implement an application as there are developers to develop it. It is important to define what is a good code quality and come up with a consistent standard that is commonly agreed by developers. One example is that a method that is written in more than a few thousand of lines of code is generally considered bad and it is of bad code quality. There are a number of attributes and requirements to look at when evaluating code quality:

- Clear design and implementation and easy to understand
- Easy to maintain
- Robust to changes
- Code duplication is minimized
- Well tested and documented

Another common way to find out code quality is through the use of cyclomatic complexity analysis. This is also the metric based analysis will be used in this project to identify code smells. Code smells are one of the main factors resulting in a poor code quality. Code smells such as long methods, duplicated code reduces the quality of the code significantly. Developers are becoming more and more aware of the importance of code quality and therefore it becomes more important to eliminate code smells and discover underlying problems. This project aims to improve the code quality of a piece of software by improving developer's ability writing good quality code.

2.2 Code Smells

As the project focuses primarily on the detection and correction of code smells, it is important to understand what code smell is. A code smell is a symptom in the source code of a program that indicates a deeper problem. It is also a warning sign in your code that leads you to uncover the underlying problem. Code smells are not bugs, they are not technically incorrect and do not prevent the code from functioning. However, the exists of code smell indicates weaknesses in design that may slow down development or increase the risk of bugs or failures in the future.[5]

There are many different kinds of code smells, some of they indicate a poor object-oriented design, some of they show a lack of proper up-front design and also many are introduced due to the lack of development time. The following sections discuss the three code smells tackled in this project in more details.

2.2.1 Using Conditional Instead of Polymorphism

It is often the case that a method should perform different tasks or behave differently according to same conditions. For a quick and dirty implementation, a switch case statement can be used to check for different conditions and operate differently for each case as illustrated below:

```
public double getSpeed() {
    switch (type) {
        case PENGUIN:
            return 1;
        case SPARROW:
            return 2 * baseSpeed;
        case PARROT:
            int totalSpeed = baseSpeed > 10 ? baseSpeed : 10;
            return totalSpeed;
        default:
            return 0;
    }
}
```

Listing 2.1: Different behavior based on a condition in switch statement

There are a number of problems associated with this approach. Firstly, there are a lot of duplicated code, since each of the conditions might be similar and for each case a similar action might be performed. Secondly, it violates the Tell-Don't-Ask principle[6] where it is recommended that instead of asking an object about its state and then performing actions based on this, it is suggested better to tell the object what it needs to do and let it decide how to do it.

As this is an obvious code smell, it should be refactored by creating subclasses matching the conditions and implementing a shared method and move the corresponding behavior into each of the subclasses. This large switch case block can then be replaced with a simple method call on the object, and the correct behavior would be performed depends on the type of the object which will result in a proper use of polymorphism as illustrated in the Listing 2.2 below.

```
abstract class Bird {
    abstract double getSpeed();
}

class Penguin extends Bird {
  double getSpeed() {
    ...
  }
}

class Sparrow extends Bird {
  double getSpeed() {
    ...
  }
}
```

```
class Parrot extends Bird {
  double getSpeed() {
    ...
  }
}
```

Listing 2.2: Use of polymorphism to replace switch statement

After the refactoring, this block of code also becomes more readable and extensible. If a future extension is required, a different kinds of bird could be added by simply adding another class extends Bird and implement the method **getSpeed()** accordingly. This extension approach does not modify existing code base and greatly increases the mitigation of introducing any regression bugs.

2.2.2 Excessive Cyclomatic Complexity

Cyclomatic complexity[7] is a quantitative measure of the number of linearly independent paths through a program's source code. It was invented by Thomas McCabe used to indicate the complexity of a program. The cyclomatic complexity is calculated by counting the number of paths flow through the code. For example, if a program contains a single IF statement with two conditions in it then there would be two different paths going through the program so its cyclomatic complexity would be 2. This is illustrated below in Listing 2.3.

```
boolean twoCyclomaticComplexity(String s) {
    if (s == null) {
        return false;
    }
    return true;
}
```

Listing 2.3: A method with cyclomatic complexity 2

One of Thomas McCabe's original application was intended to reduce the cyclomatic complexity in a program and he recommended developers to count the cyclomatic complexity of the program they are developing and try to split the larger module into small modules with cyclomatic complexity less than or equal to 10 as suggested in his book "Structured testing"[8].

There are studies finding that there is a positive correlation between the cyclomatic complexity of a program and the number of defects contained. Although larger applications tend to have higher cyclomatic complexity and have more defects due to the fact that it has more lines of code, the size of the application is not controllable in commercial software development, so this has not being proven and accepted by commercial software development organizations.[9]

However, a program with excessive cyclomatic complexity normally indicates that it is "smelly" and more complicated than it is supposed to be, and it could be split into smaller modules using different refactoring techniques. In addition, programs with low cyclomatic complexity tends to be quicker to understand by human and easier to achieve higher test coverage rate with increasing testability.

2.2.3 Return Private Mutable Field

In Java, there are mutable and immutable objects. Immutable objects are ones that do not allow alternation to its publicly available states after they are instantiated, they give a guarantee that its state would not change over the lifetime of the object. The String class[10] in Java standard library is a good example of a immutable object. On the other hand, a mutable object allows modification to its states during its lifetime in the program, so there would not be any guarantee on its state.

There are many advantages using immutable objects in Java programs, the most important benefit being that it simplifies programming. Using immutable objects, programmers can be sure about the state of an object as long as it is properly constructed without considering when it was instantiated or what are the side effects caused on this object, and never worry that it gets into an inconsistent state.

With all the benefits using immutable objects, the problematic of mutable objects are also exposed. One of the issues mutable objects could cause is when they are being used as class members. It is nowadays a standard practice to define the access of class members private in order to restrict access to some of the object's internal states for encapsulation purpose. In order to allow public access to some of these class members, "getter" methods are used. A "getter" method is a method that is used to get a private class field but not exposing the value contained in the field. An example of a "getter" method is illustrated in Listing 2.4 below.

```
class MyClass {
    private String name;

    public MyClass() {
        this.name = "immutable string";
    }

    public String getName() {
        return s;
    }
}
```

Listing 2.4: Getter method example

A "getter" method seems straightforward and flawless at the first glance. However, it could be problematic and vulnerable to the whole program if it is not written with extra care. The code snippet in Listing 2.5 below demonstrates how a "getter" method could cause problems.

```
class MyClass {
  private Date date;

  public MutableClass() {
    this.date = new Date();
  }

  public Date getDate() {
    return date;
  }
}
```

Listing 2.5: Getter method returns private mutable field

In the code snippet above, although the class field **date** is declared as private level access, using the **getDate()** method could still obtain the reference to the date Object and then its state could then be altered by setting its time as below:

```
date.getTime().setHour(5);
```

Listing 2.6: Mutate private mutable field value

In order to prevent this from happening, the **getDate()** method should first create a clone of the date and then return the cloned object rather than the real class member. This would prevent the caller of the **getDate()** method gets hold of the reference to the private class field and later alter its state.

Return reference to private mutable data is problematic, but developers are very easy to miss it and it does not cause any compiler warnings. It could not only cause unexpected behavior when a program is running but also lead to a series of vulnerabilities in the application. This project aims to implement an automated detection of this issue as part of the IntelliJ plugin and suggest refactoring guidance to the developers to fix the problem.

2.3 Existing Tools

2.3.1 Checkstyle

"Checkstyle is a static code analysis tool used in software development for checking if Java source code complies with coding rules." [11] This is the description of the Checkstyle tool described in Wikipedia. Checkstyle[1] is one of the mainstream open source static code analysis tools available for developers intended to help programmers write Java code that adheres to a coding standard. Checkstyle was written in Java in 2011 originally available as Ant task and command line tool and later both IntelliJ plugin and Eclipse plugin are developed around Checkstyle to provide real-time feedback to developers.

There are different modules in Checkstyle, each of them is responsible for analysis and detection of different problems ranging from Java docs, naming conventions to class imports and functions. One of the biggest advantages using Checkstyle is that it is highly configurable and each module can be configured to support different coding standards by supplying customized configuration files. This allows different development teams or sole developers to define a set of rules they would like to follow and this greatly helps them to comply with the same set of coding standards and practices during development which would result in the improvement of code quality and reduce the development cost. Although Checkstyle provides a complete set of checking for coding styles as well as detection of local issues in Java code, it does not provide functionality to analyze design problems in application level or architectural level. In addition, Checkstyle detects problems and alerts developers to fix the problems but it does not provide fix suggestions for the detected problems. Experience and expertise from the developers are required to modify the program to comply the rules.

2.3.2 FindBugs

FindBugs is another open source static code analysis tool widely available for developers in the community. As its name suggests, it focuses on "using static analysis to look for bugs in Java code"[12]. FindBugs was created by Bill Pugh and David Hovemeyer and it was first released in 2006. FindBugs was originally developed as a stand-alone GUI application and later available as plugins for a wide range of IDEs including Eclipse and IntelliJ IDEA. It also has a successor

SpotBugs which was described as “the spiritual successor of FindBugs, carrying on from the point where it left off with support of its community” in its official website [13].

Compared to Checkstyle, FindBugs focuses on detecting bugs in the program rather than coding styles, it also operates on Java bytecode instead of the source code. The analysis on Java bytecode means that it requires compilation of the Java program which might be time consuming and does not provide a short feedback loop for the programmers.

2.3.3 JDeodorant

“JDeodorant is an Eclipse plug-in that detects design problems in Java software, known as code smells, and recommends appropriate refactorings to resolve them.”[14] JDeodorant currently supports the detection of five different code smells - Feature Envy, Type Checking, Long Method, God Class and Duplicated Code. Similar to the aim of this project, JDeodorant also provides fix suggestions after the detection of the code smells.

There are a number of differences between JDeodorant and this project in addition to the difference in the code smells being detected. The first being that JDeodorant is an Eclipse plugin which has a very different ecosystem from the IntelliJ plugin this project aims to develop. Secondly, JDeodorant conducts analysis of the program as a whole which takes a long time, and the analysis is only triggered when developer invokes it manually.

This project on the other hand tries to detect code smells while a developer is working on development tasks in order to minimize the feedback loop. Step by step refactoring guidance is also provided to the developers to assist in the code smell refactoring process.

2.3.4 Vignelli

Vignelli is an IntelliJ plugin used to improve software design by speeding up the software design feedback loop. Similar to JDeodorant, Vignelli is developed as an IDE plugin directly and focuses on a set of problems in a similar level. It focuses on three different design problems below:

1. Train wrecks
2. Direct uses of singletons
3. Long methods

The main difference between Vignelli and other existing code analysis tools such as Checkstyle or JDeodorant is that it tries to provide quick feedback loop to the developers which is also the aim of this project. According to the Vignelli paper[15], this is achieved by performing partial analysis of the abstract syntax tree of the program in real time with approximate analysis time under average one milliseconds. This is what sets Vignelli apart from other existing code analysis tools.

In order to provide a step by step refactoring suggestions to the developers, Vignelli does not provide a “Quick fix”[16] option but instead designs a set of user interface to explain the required changes to the developer and requires a series of user inputs in order to follow the refactoring steps. As stated in the Vignelli report, “Vignelli successfully assists in the refactoring of 50% of “train wrecks” and 73.68% of “direct uses of singletons”.”

This approach might benefit developers who are not familiar with the refactoring steps but it potentially slows down the refactoring process and decreases the usability of the tool. As well as providing a step by step refactoring suggestions, this project makes use of the "Quick Fix" option to allow users to enter the refactoring process easily and conveniently in order to provide them a smooth user experience.

2.4 IDE Plugin and The Java Language

One of the objectives in this project is to develop a tool that would benefit a wide range of developers. In order to achieve this objective, this tool should provide code analysis for a popular language so that it can be used widely in the community. According to the Stack Overflow Developer Survey result 2017[17], Java has been ranked the third most popular technology in the Stack Overflow community while the first and second being Javascript and SQL respectively. There are a lot of different kinds of design problems and code smells in Java programs which makes an automated code analysis tool even more important for Java developers.

In order to make the tool easy and convenient to use by Java developers, it should be easy to obtain, convenient to set up and configure as well as fitting into part of the development routing of applications. An IntelliJ plugin would fit for this purpose precisely. Firstly, IntelliJ IDEA[18] is specifically designed to provide a complete set of support for Java application development, it features an out-of-the-box user interface and strives to provide a maximum developer productivity. In addition, it facilitates powerful static analysis support for third party plugins which would allow this project to focus on the implementation of code smell detection and correction, and develop on top of the provided APIs including Abstract Syntax Tree (AST) and Program Structure Interface (PSI)[19] (see Section 3.1.1).

Furthermore, an IntelliJ plugin would be able to provide a fast feedback loop to the developers while they develop the application compared to stand-alone application or command line tools. Developers could get warning and notifications while they code and before they finish the program. Once they are alerted and realized that the code smell exists in the program, they could follow step by step refactoring instructions within the IntelliJ platform to fix the problem. All the steps can be completed within IntelliJ IDEA, the developer does not need to leave the IntelliJ window to perform any actions which provides a smooth user experience, and greatly increases the usability of this tool.

2.5 Code Inspection

Code inspection is often referred to "Fagan inspection" after Michael Fagan, who creates a very popular software inspection process[20]. The inspection is a structured process used to find defects in the source code of a piece of software and it is normally used during different phases of the software development process.

In this project, we aim to develop a tool that analysis code automatically and therefore it can be classified as a code inspection tool. However, it is important to note that there are two different approaches for code inspection tools:

1. The tool automates the code inspection process, making developers easier to follow the guidelines and record the results.
2. The tool performs automatic code inspection, relieving the programmers of the manual inspection burden.

The tool we would like to develop in this project features a mixture of the two approaches. It tries to automate the code inspection process so that developers can focus on the actual development task without performing manual inspections, but it also performs this code inspection automatically in the background as the developer is making changes to the code which further improves developer's productivity by shortening the feedback loop. It is important to realize the advantages gained by combining the two approaches together and this feature on its own distinguishes the tool developed in this project from other existing code inspection tools.

Chapter 3

Design and Implementation

In this project we develop Automatic Code Smell Detector, an IntelliJ IDEA plugin that continuously inspects code that a developer is working on, detects potential code smells, and provides intuitive user interface to guide developers through the refactoring process.

3.1 Software Architecture

The plugin is largely organized in two parts: 1) the detection part which is responsible for code inspections and examinations in order to identify code smells; 2) the correction part which assists developers in the code smell refactoring process. Each of the two parts is made up of a number of components as shown in Figure 3.1. We will first discuss the two parts respectively and each of the smaller components will be explained in more detail in the following sections.

The detection part of the plugin is built on top of the IntelliJ Inspections System which provides a number of features that are convenient to use for developing custom code inspections, help to speed up the overall plugin development, and ensure a consistent user experience with other built-in inspections. Firstly, the code inspections are run in background thread without affecting user's interaction with the IDEA, this out-of-the-box feature hides away a lot of the complexities when dealing with low level logic handling worker threads and UI threads, and allows more attention to be put on the actual code smell detection. Secondly, code inspections are automatically triggered on every code changes made, different optimizations are provided by the Inspections System to improve the code inspection performance. For example, the Local Inspection Tool ensures that inspections are only run on the current opened file instead of an inspection on the project level. Another optimization is that code inspections are only rerun on the code blocks that has changed. These strategies largely shorten the feedback loop and developers are notified about any inspection results within seconds. Thirdly, each of the code smell inspections in the plugin makes use of the code highlighting feature provided. It selects the code block which contains a code smell and renders a standard warning highlighting to warn the user. In addition, by extending the Inspections System, developers can manage code smell inspections together with other build-in code inspections all in one place, as well as having the options to suppress warnings, configure inspection severity, and disable individual inspection etc.

There are pros and cons building the code smell detection around IntelliJ Inspections System. One of the biggest disadvantages is that the plugin is heavily platform dependent. Since the code inspections make use of a lot of utilities provided by the IntelliJ Inspections System, the plugin is therefore hardly portable to other IDEs such as Eclipse or Visual Studio as they have a very different architecture. Another drawback is that the code smell inspections are less flexible and it is very difficult to apply any bespoke optimization strategy to improve the code inspections performance. For example, it is not possible to prevent the code inspections to rerun against code changes even if there might be heuristics to determine that the code changes are not relevant.

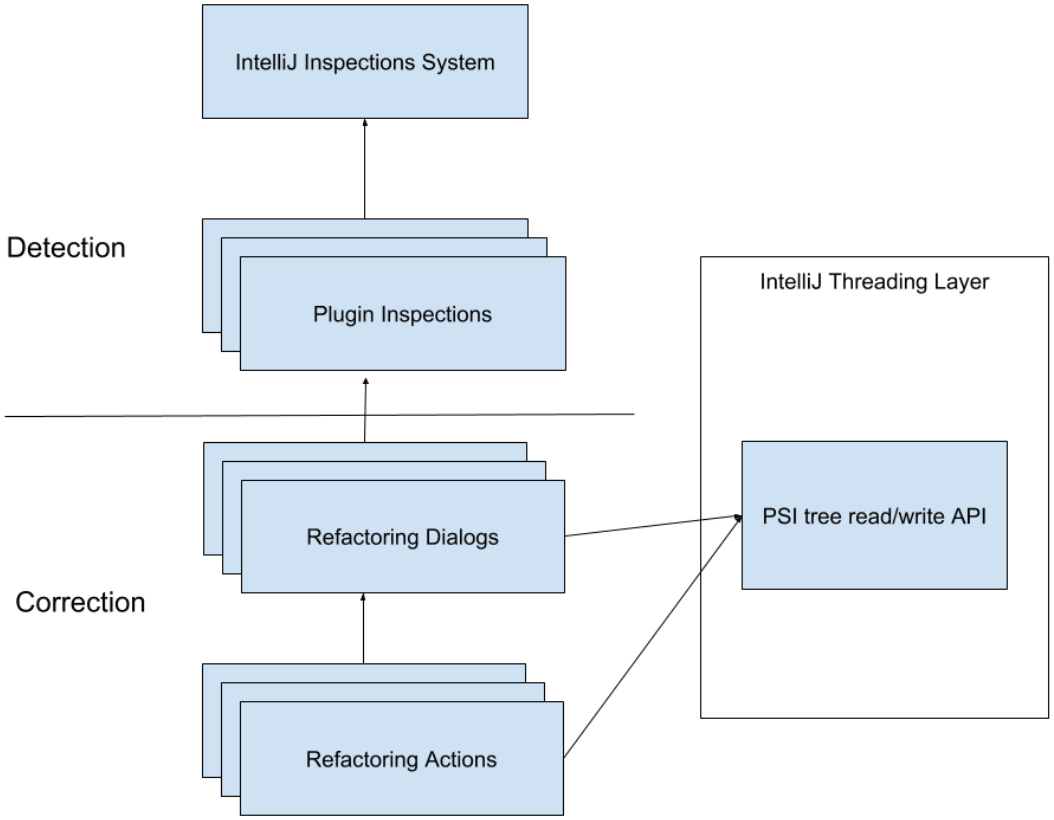

Figure 3.1: Plugin Architecture Overview

On the other hand, the correction part of the plugin is made up of two main components: the refactoring dialogs and the refactoring actions. These two components combined to provide users step by step refactoring guidance and assist them to remove the previously found code smells. Unlike the detection part of the plugin, the correction part involves both analyzing and modifying existing code which are conducted through the Program Structure Interface(PSI) layer exposed by the IntelliJ platform.

For each of the code inspections, a correction strategy is associated with it through the "Quick Fix" configuration. The correction phase of the plugin begins when users enter the "Quick Fix" mode of the found code smell. The code block that contains the code smell is identified and its PSI representation is passed into the correction component from the detection component. The PSI element is then reviewed and based on the type of code smell it contains, a specific refactoring dialog provider will be used to generate a series of refactoring dialogs to guide users through the refactoring process. During each of the refactoring steps, one or more refactoring actions are used to make changes to the PSI tree. Some of these refactoring actions are provided by the IntelliJ platform out of the box. For example, the built-in extract method refactoring action takes a code fragment that can be grouped together, move it into a new method and replace the old code with call to the new method. Other actions that are more specific to code smell refactoring are implemented specifically (see Section 3.1.6).

3.1.1 Program Structure Interface (PSI)

At the core of the IntelliJ Platform, Program Structure Interface(PSI) is the layer which is responsible for parsing files, generating syntactic and semantic code models, and representing the internal hierarchical structure of source code [19]. Individual PSI elements are used to explore the internal structure of project as it is interpreted by the IntelliJ Platform. A hierarchy of individual PSI elements forms a PSI tree, there is one or more PSI trees contained in a PSI file. A PSI

file is the root of a structure representing the contents of a file, while itself being a PSI element. Figure 3.2 below shows an example of a PSI file which contains a PSI class which further contains a number of PSI methods. Each element provides direct access to its parent element or its children elements, which makes navigating from both top to bottom and bottom to top very convenient. Access to sibling elements is also available which further benefits the navigation between elements. One of the most important use cases of PSI element is performing code analysis, such as code inspections or refactoring actions. This hierarchical structure of PSI elements exactly provides great efficiency and performance advantages when analyzing code structures and conducting code smell inspections.

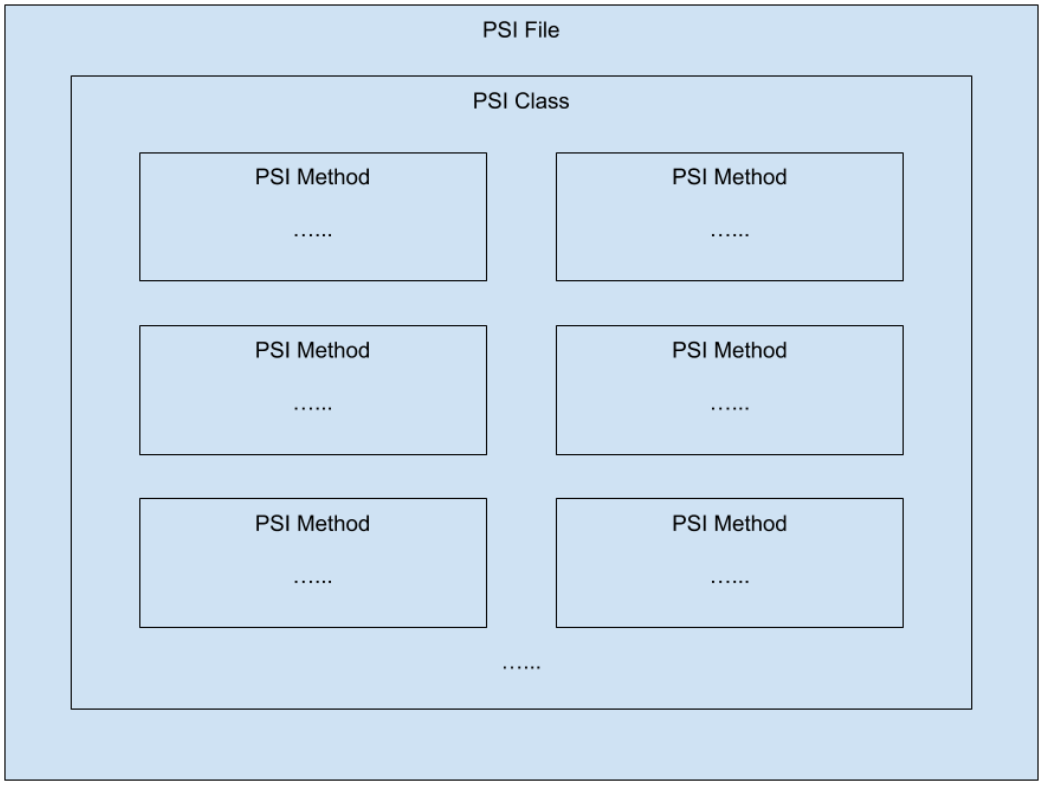

Figure 3.2: PSI File example

In this project, in addition to the general PSI elements (PSI class, PSI method etc), we also work with some of the Java PSI elements which is a language-specific extension to the PSI layer. The Java PSI elements each contains Java specific features. For example, the PSI Java file features APIs including *getLanguageLevel()* which returns a specific Java version used in a file, and *getPackageName()* which returns the name of the package to which the file belongs.

In addition to the language specific APIs, PSI elements also provide a rich set of general APIs to allow element creation, modification, deletion and data access. During the detection phase of the plugin, we make use of the data access APIs to observe the source code structure, conduct metric based analysis and identify code smell patterns within code blocks. After a code smell is identified, we then modify existing elements, update the PSI tree structure, and create new PSI elements. For example, the refactoring process of the using conditional instead of polymorphism code smell involves creating new PSI classes, modify existing PSI methods and removing some of the PSI elements.

Although we make use of these APIs during the refactoring process, we implement an intermediate layer called "Refactoring Actions" which handles most of the direct usage of these APIs (see Figure 3.3). Actions make consecutive calls to the PSI element APIs and all these operations on the PSI element constitute to a single refactoring action. For example, the extract method

action makes use of different APIs provided by the PSI element to create new PSI method, add the created method to the PSI tree and move the target code block into the new method. This additional layer offers several benefits. Firstly, some of these actions perform common operations to PSI elements and therefore can be easily reused by other refactoring steps. In addition, concurrency controls could be enforced within the actions to prevent race conditions caused when a PSI element is modified by more than one thread at the same time. This hides the low level threading logic away from the refactoring steps, which could focus on perform actual code refactoring.

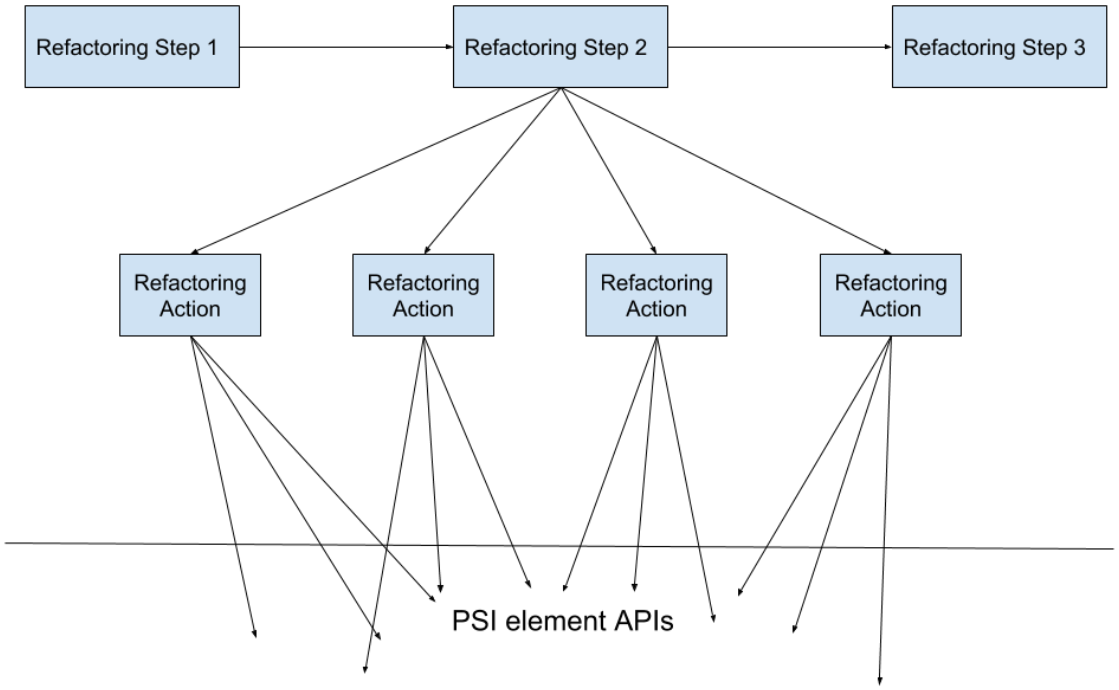

Figure 3.3: Refactoring Actions Layer

3.1.2 IntelliJ Threading Layer

In the IntelliJ Platform, there are many different code-related data structures, the Program Structure Interface(PSI) we discussed above is one example. Most of these data structures can be read from or write to by different threads and processes within IntelliJ IDEA. IntelliJ Platform defines general threading rules [21] to ensure that no race conditions could occur when performing write operations to data structure, users actions are not blocked by running background tasks, and UI is prevented from freezes when read actions take a long time. In order to comply with these threading rules, a number of APIs are provided for plugin developers to use. For example, *ApplicationManager.getApplication().runReadAction()* must be used to wrap read actions when they are performed outside UI thread.

Throughout the plugin, we make full use of these wrappers to ensure all the read and write operations comply with the IntelliJ general threading rules. While these wrappers make sure threading rules are followed, there are several downsides. The introduction of these wrappers brings a lot of complexity into the plugin source code, as these boilerplate code are all over the place and difficult to avoid. The approach being used to minimize the complexity is to isolate the boilerplates with other refactoring logic - we try to move them into refactoring actions so that they are hidden from higher abstractions. Another downside is that the learning curve for developers

who are not familiar with IntelliJ plugin development is significantly increased.

It is hoped that the IntelliJ development team would consider to improve this mechanism, so that the threading complexity is hidden from plugin developers.

3.1.3 Code Inspections

The entry point of the plugin is the code inspection component. Each code inspection component corresponds to one of the code smells in this project and is responsible for the identification of the specific code smell. As shown in Figure 3.4 below, the code inspection component extends from the IntelliJ Inspections System in order to take all the advantages of IntelliJ built-in code inspections utilities and ensure that developers are able to use the code inspections the same way as the built-in ones. We will first discuss the internal structure of the code inspection component and follow by an overview of how it interacts with other components in the system.

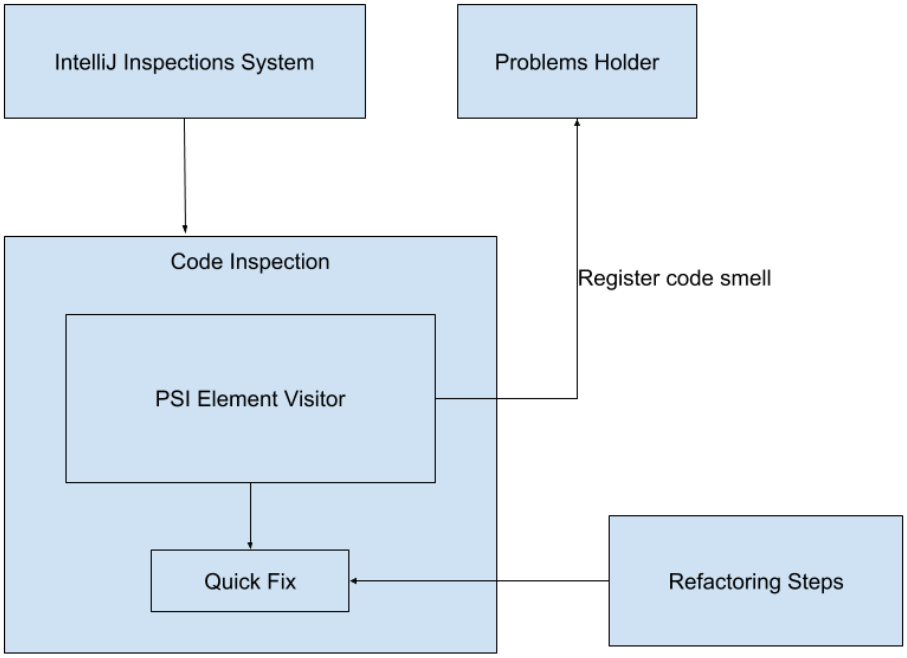

Figure 3.4: Code inspections structure overview

IntelliJ Platform provides a *LocalInspectionTool* API to allow custom code inspections to build on top of the IntelliJ Inspections System. Listing 3.3 shows the *ReplaceCondtionalWithPolymorphism* code inspection extends from the *BaseJavaLocalInspectionTool* class.

```
public class ReplaceConditionalWithPolymorphismInspection
        extends BaseJavaLocalInspectionTool {

    ......

}
```

Listing 3.1: Example of creating dialog UI programmatically

By extending the *BaseJavaLocalInspectionTool* class, it allows the code smell inspection to specify its displayed name, the inspection group to which it belongs, and whether the inspection is enabled by default etc. All of these are configured by overwriting some of the methods from super class as shown in Listing 3.2 below.

```
@Override
@NotNull
public String getDisplayName() {
    return "Replace conditional with polymorphism";
}

@Override
@NotNull
public String getGroupDisplayName() {
    return CODE_SMELL;
}

@Override
public boolean isEnabledByDefault() {
    return true;
}
```

Listing 3.2: Configure basic settings by override super class methods

At the core of the code inspection, the PSI element visitor contains most of the logic. We implement the PSI element visitor to visit the PSI tree generated from the code that a developer is working on and traverse recursively down the tree. The PSI element visitor observes patterns in the source code and tries to detect a specific code smell. The visitor normally focuses on the analysis of a subset of PSI elements in the PSI tree. For example, in order to detect the excessive cyclomatic complexity code smell, the visitor examines each PSI method in the PSI tree and uses metric based analysis to calculate the cyclomatic complexity within the method. Once a code smell is detected, the code smell is registered with a Problems Holder provided by the IntelliJ Platform together with a short description of the code smell.

The Problems Holder serves as a common place where different code problems detected are collected together. The exists of the Problems Holder is important because it ensures that code problems reported by different plugins are dealt with in a consistent way including notifying users with code highlighting, providing a short description of the code problem, and allowing users to perform different options such as suppressing warnings or applying quick fix.

Another important component within the code inspection component is the "Quick fix". Quick fix is a mechanism provided by the IntelliJ Inspections System as a way to provide a one step fix for general code problems. In this project, we make use of the quick fix option to allow users to enter the refactoring process. Once a code smell is found, not only itself is reported to the Problems Holder, the customized quick fix option is also associated with the reported code smell.

3.1.4 Refactoring Dialogs Provider

In the plugin architecture diagram above(see Figure 3.1), refactoring dialogs are directly used in the correction phases of plugin. Although this is true from a user's point of view, from the technical aspect there is an additional layer - the refactoring dialogs provider which sits in front of the refactoring dialogs as shown in Figure 3.5.

As the name suggests, the refactoring dialogs provider is used to generate refactoring dialogs. There is a series of refactoring dialogs involved in the refactoring process of a single code smell,

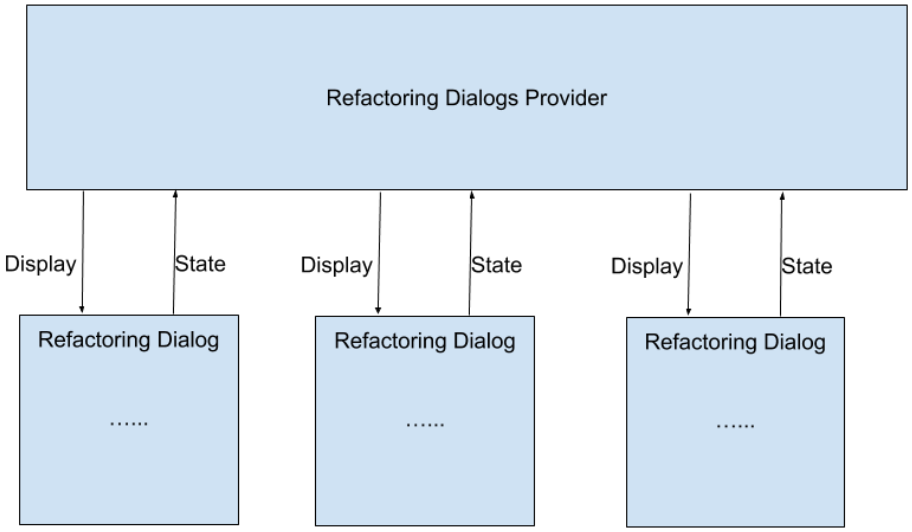

Figure 3.5: Refactoring Dialogs Provider Overview

as we aim to provide a step by step guidance to the developers. It is necessary to manage these dialogs all in one place and provides an unified API to display them. The refactoring dialogs provider exists to provide an abstraction to simplify the creation of the dialogs. It not only hides away the complexity of generating the dialogs, but also manages the state of the dialogs. For example, the plugin needs to know the state of the dialog and whether the user closes the dialog, we include these state management logic within the refactoring dialogs provider so that they can be accessed all in one place. We implement a refactoring dialogs provider for each of the code smells in this project, and the refactoring dialogs are grouped under the same provider based on which code smell they target at.

3.1.5 Refactoring Dialogs

The refactoring dialogs are one of the most important elements in the code smell refactoring process. During the refactoring process, each step is represented by a refactoring dialog. The dialog presents information about the code smell, describes the refactoring to be done at the current step, and allows the developer to continue to the next step or cancel the refactoring.

There are two approaches to create the dialog UI in the plugin and both approaches make use of the DialogWrapper [22] provided by the IntelliJ Platform. We have used both approaches throughout the project depending on the purpose of the dialog and with careful consideration of its props and cons.

One of the approaches is to use the UI Designer forms by binding the form to the class extending DialogWrapper. Figure 3.6 shows an example of the UI Designer form.

Using the UI Designer form has a number of benefits. Firstly, it helps to speed up the UI design task significantly. Because it provides a seamless integration with the Swing library, Swing

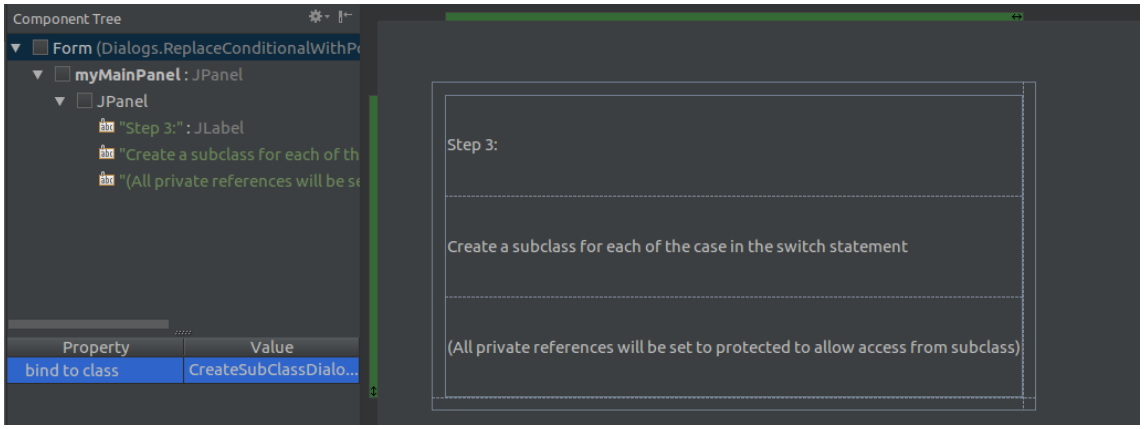

Figure 3.6: UI Designer form example

components can be created by drag and drop without writing any UI code, and adjusting the position of a component is simply dragging the component to the target position. Secondly, the UI visualization powered by the IntelliJ IDEA allows the UI to be reviewed without compiling and running the plugin and any UI issues are noticed immediately. This shortens the feedback loop during the plugin development and further speeds up the UI design task. Last but not least, the use of UI Designer form isolates the UI code from any other logic used to interact with the UI.

However, there are also limitations come with all the benefits. The UI Designer form is very easy to use when adding simple Swing components, on the other hand, it is difficult to have fine-grained control on the style of the components. In addition, the UI Designer form is rendered in IntelliJ IDEA based on a XML file. When we create an UI Designer form, the XML file is automatically generated. Any changes made to the UI Designer form are then eventually converted to the modification of the XML file. The auto-generated code introduces the problem of readability and makes it nearly impossible to understand and edit the underlying XML file directly.

The second approach we used to create the dialog UI is writing Java Swing code directly. Instead of binding a UI Designer form to the class extending DialogWrapper, we create Swing components like JPanel programmatically within the class (see Listing 3.3). We use this approach when we want to have precise control of the style of the Swing components or generate dynamic contents in the dialog. For example, when we want to display the code block being extracted within the extract method dialog, implementing the UI programmatically would give us a lot of flexibility and provides fine-grained control on the style of the code block displayed. Writing Swing code directly not only provides extra flexibility compared to using UI Designer form, but also makes the UI code more readable.

```
@Override
protected JComponent createNorthPanel() {
    JPanel panel = new JPanel(new GridBagLayout());
    GridBagConstraints gbc = new GridBagConstraints();
    gbc.anchor = GridBagConstraints.WEST;
    gbc.fill = GridBagConstraints.BOTH;
    gbc.insets = JBUI.insets(4, 0, 4, 4);

    //first line
    gbc.gridwidth = 1;
    gbc.gridx = 0;
    gbc.gridy = 0;
    panel.add(new JLabel("Step 6:"), gbc);

    //second line
    gbc.gridy++;
```

```
        panel.add(new JLabel("The method containing switch " +
                "statement is made abstract"), gbc);

        //third line
        gbc.gridy++;
        panel.add(new JLabel("This class is also made abstract"), gbc);

        return panel;
    }
```

Listing 3.3: Example of creating dialog UI programmatically

One of the aims in this project is to assist developers to go through the refactoring process step by step. In order to accomplish the goal, it is worth to mention that we introduce an interesting structural characteristic in the refactoring dialogs, which is the "refactoring dialogs chain".

As discussed earlier, each refactoring dialog represents a single refactoring step. A step by step refactoring process naturally chains the dialogs together. Therefore, we implement an Interface that enforces this characteristic (see Listing 3.4). Refactoring actions implement this interface and override the two methods with their own implementation. The *performAction()* method makes calls to different refactoring actions to perform the necessary refactoring in the current step. Note that this method returns a *boolean* which is used to indicate if the refactoring has been performed successfully or canceled. If the refactoring is performed successfully, the user should continue to the next refactoring step which should be triggered in the *performNextStep()* method, as such the next refactoring dialog is chained to the current one. This forms a chain of refactoring dialogs as shown in Figure 3.7 below.

```
public interface ChainedDialog {

    public boolean performAction(PsiElement element);

    public void performNextStep();

}
```

Listing 3.4: ChainedDialog Interface

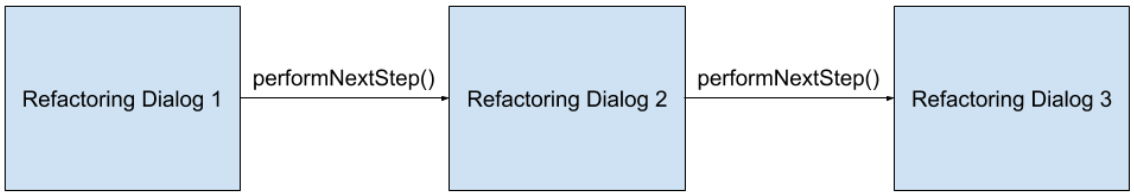

Figure 3.7: A chain of refactoring dialogs example

3.1.6 Refactoring Actions

Refactoring actions are another set of components in the correction part of the plugin. They are generally used within the *performAction()* method we discussed above. Each refactoring action is

responsible for a specific operation to a PSI element by using a combination of the PSI element APIs.

The PSI element that is modified by the refactoring action comes from the detection phase of the plugin. During the detection phase, a code block that contains a code smell is identified. Then the PSI element that represents that code block is extracted from the PSI tree and passed to the refactoring process. During each refactoring step, changes are made to a subset of the PSI element using the refactoring actions.

Although one or more refactoring actions are associated with each refactoring step, it is worth noting that sometimes the refactoring actions might not be performed. In this project, we aim to provide step by step refactoring guidance, this is achieved by ensuring the source code is in the intended structure before proceeding to the next step. Therefore, when the source code is already in the shape of form that is sufficient to proceed to next step, no refactoring action would be required. The flowchart shown in Figure 3.8 illustrates the control flow between two refactoring steps. It is obvious to see that the refactoring action is performed only when modification to the source code structure is required, and the refactoring process proceeds to the next step after the refactoring action being performed successfully.

```
public int getSpeed() {
    if (baseSpeed < 0) {
        return 0;
    }
    switch (type) {
        case PENGUIN:
            return 1;
        case EUROPEAN:
            return 2 * baseSpeed;
        case NORWEGIAN_BLUE:
            int totalSpeed = baseSpeed > 10 ? baseSpeed : 10;
            return totalSpeed;
        default:
                return 0;
    }
}
```

Listing 3.5: Example of method contains a if statement and a switch statement

To be specific, we present the refactoring procedures of the using conditional with polymorphism code smell, apply to two different code blocks and compare their differences. The first example is shown in Listing 3.5. The first step in the refactoring is to ensure that the switch statement is the only statement in a method. In this case, the code is not in the intended structure, so it would be necessary to perform the extract method action to extract the switch statement into a separate method. The refactoring process only proceeds to the next step after the refactoring action taking place. On the other hand, the method in Listing 3.6 contains only a single switch statement. Therefore, the extract method action can be skipped and the refactoring process proceeds to the next step.

```
public int getSpeed() {
    switch (type) {
        case PENGUIN:
            return 1;
        case EUROPEAN:
            return 2 * baseSpeed;
        case NORWEGIAN_BLUE:
            int totalSpeed = baseSpeed > 10 ? baseSpeed : 10;
```

```
            return totalSpeed;
        default:
                return 0;
    }
}
```

Listing 3.6: Example of method contains only a switch statement

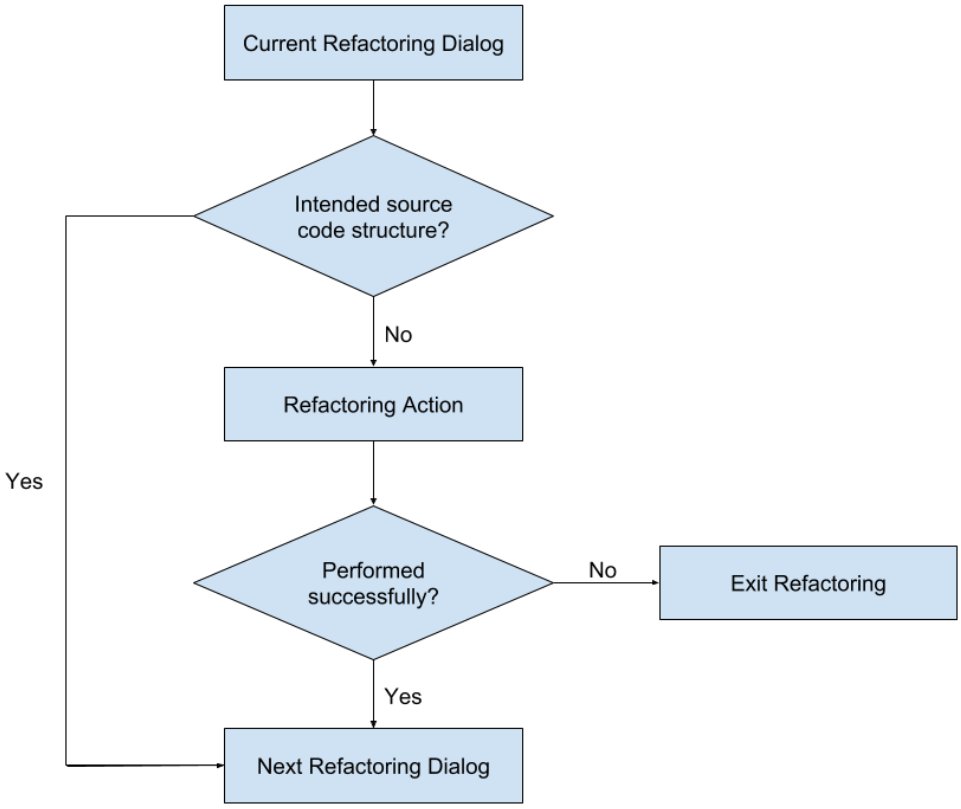

Figure 3.8: Refactoring process flowchart

3.2 User Interface Design

In order to provide a seamless experience to developers while they use the plugin, it is important to design user interface with the following goal: 1) The UI should be easy to understand and self-explanatory. Developers should be able to follow the refactoring process intuitively without any specific instructions; 2) The UI should be consistent with other components in IntelliJ so as to eliminate any confusions developers might have and minimize developer's learning effort; 3) The plugin overall should not distract developers from performing their coding task while being easily accessible when code smell is detected.

3.2.1 Plugin Enabling and Disabling

The plugin provides a number of code smell detections and corrections, some of which might not be useful or relevant to all developers. By using the code inspection system in IntelliJ, we provide an intuitive way for developers to enable and disable one or more code smell inspections easily

(see Figure 3.9). Within the IntelliJ Inspections settings page, developers could manage not only the code smell inspections but also other code inspections provided by IntelliJ. The screen-shot shows the code smell section of the IntelliJ Inspections settings page. By ticking the check-box at the end of each code smell, developers could decide which code smell inspections to be run in the background. Developers could also enable or disable all the code smells all together using the check-box next to the highlighted "Code Smell" section.

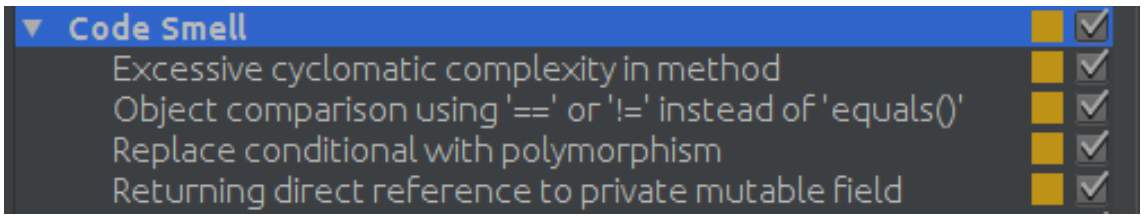

Figure 3.9: Code inspection settings

3.2.2 Code Smell Highlighting

During the detection phase of the plugin, inspections are run in the background against any code that a developer is working on, developers do not have to perform any action and can focus on their coding task. The developer is only notified once any code smell is found. In order to warn the developer that there is a code smell found, the plugin highlights the block of code that contains the code smell using standard warning highlighting in IntelliJ (see Figure 3.10) In this example, a potential code smell "using conditional instead of polymorphism" is identified, and the corresponding switch statement is highlighted.

As the plugin is built on top of the code inspection system in IntelliJ, there are a number of advantages offered out of the box. The first being that this is the standard highlighting for warning also used by other code inspections provided by IntelliJ, developers should be familiar with the meaning of the highlighting and there is no further instruction required to explain the intention.

Another advantage is that there are different options provided to developers out-of-the-box and they can decide which action to take. For developers who are not familiar with this code smell, they could choose to read about a short description of the code smell (see Figure 3.11). Another option is to use the so called "Quick Fix" option which will start the refactoring process (see Section 3.2.3). Developers could also have a fine-grained control on which code smell warnings to display. If they think this code smell can be ignored and do not want to be disturbed by the warning, they could simply suppress the warning for the specific piece of code. Further, if they are not interested in this code smell at all, they could disable the inspection while keeping other code inspections enabled.

```
public int getSpeed() {
    switch (type) {
        case PENGUIN:
            return 1;
        case EUROPEAN:
            return 2 * baseSpeed;
        case PARROT:
            int totalSpeed = baseSpeed > 10 ? baseSpeed : 10;
            return totalSpeed;
        default:
            return 0;
    }
}
```

Figure 3.10: Code smell warning highlighting

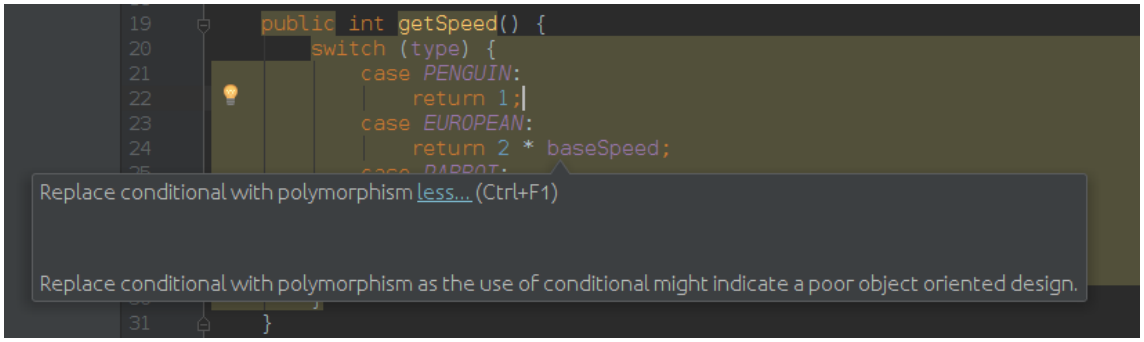

Figure 3.11: Code smell warning highlighting with description

3.2.3 Refactoring Dialogs

As mentioned above, the goal of the UI design is to ensure seamless experience for developers during their use of the plugin. We adhere to the same principle when we design the refactoring dialogs.

After a code smell is detected, the piece of code that contains the code smell is highlighted as shown previously. Users enter the refactoring process by choosing the so called "Quick Fix" option available to them (see Figure 3.12).

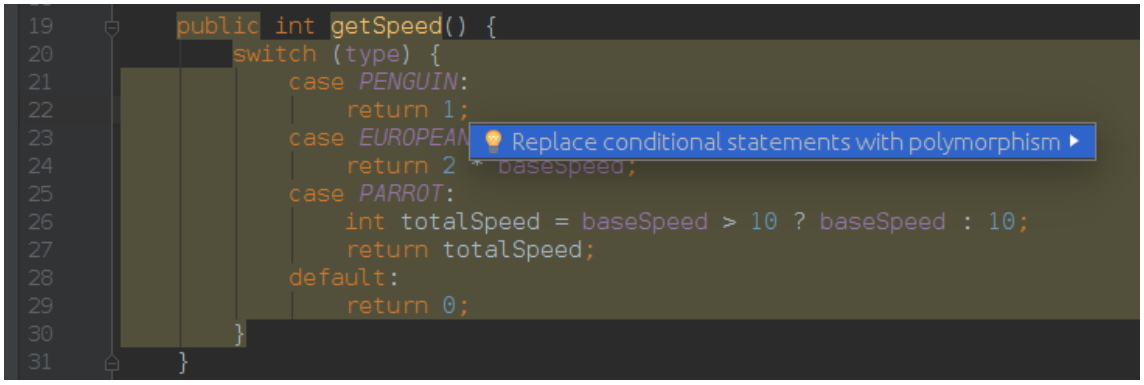

Figure 3.12: Code smell Quick Fix option

For many of the built-in IntelliJ code inspections, the "Quick Fix" option performs an one step refactoring to the selected code. For example, the quick fix for a misspelled method name would simply be correcting the spelling of the method name. In this project, the plugin intends to guide developers through the refactoring process by providing step by step guidance. This way developers are more familiar with the refactoring steps and eventually for them to learn different refactoring techniques through the process and improve their refactoring skills.

A number of dialogs would be displayed as developers go through each step of the refactoring process. The first dialog shown at the start of the refactoring process is a summary of the code smell. It explains to the user what the code smell is, why it would be necessary to remove the code smell, and the benefit of fixing the code smell. After being familiar with this code smell, users could dismiss the start refactoring dialog and no longer sees it again. Figure 3.13 below gives an example of the start refactoring dialog.

The refactoring dialog for one of the steps to replace conditional code smell with polymorphism is shown below in Figure 3.14. The dialog displays several pieces of information to users, including which step this is in the refactoring process, what is the goal to achieve in this step and what is the action to perform. Two options are also available to users. Users could choose to follow the refactoring guidance by clicking the "Next Step" button, the plugin would apply appropriate refactoring technique to the code, make necessary changes, and continue to the next step. More

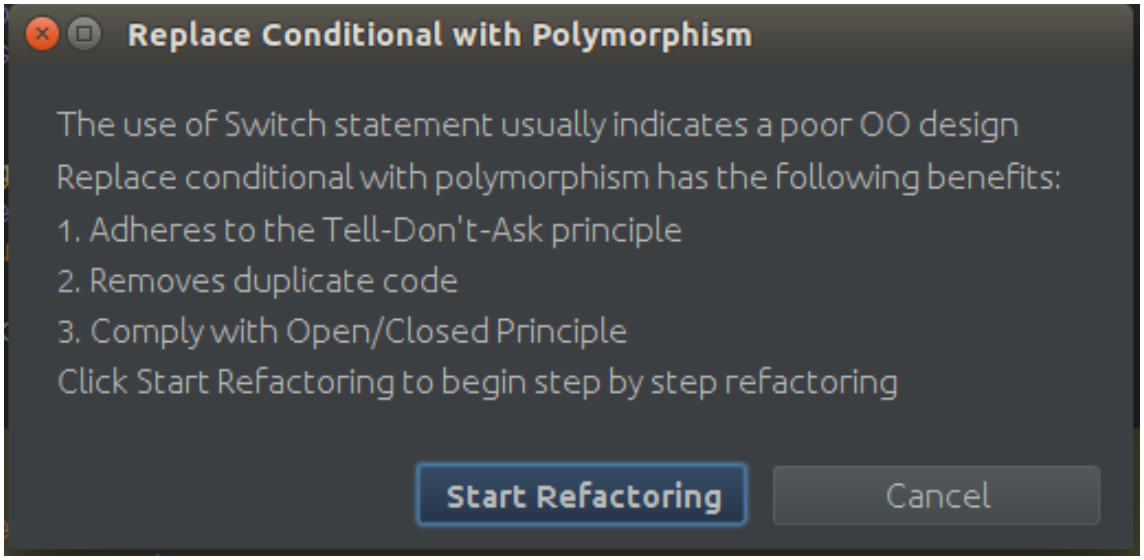

Figure 3.13: Start refactoring dialog

experienced developers, however, could decide to not follow the current step and instead performing their own refactoring. This provides great flexibility to the users and allows developers with different levels of coding experience to make use of the plugin and benefit from using it.

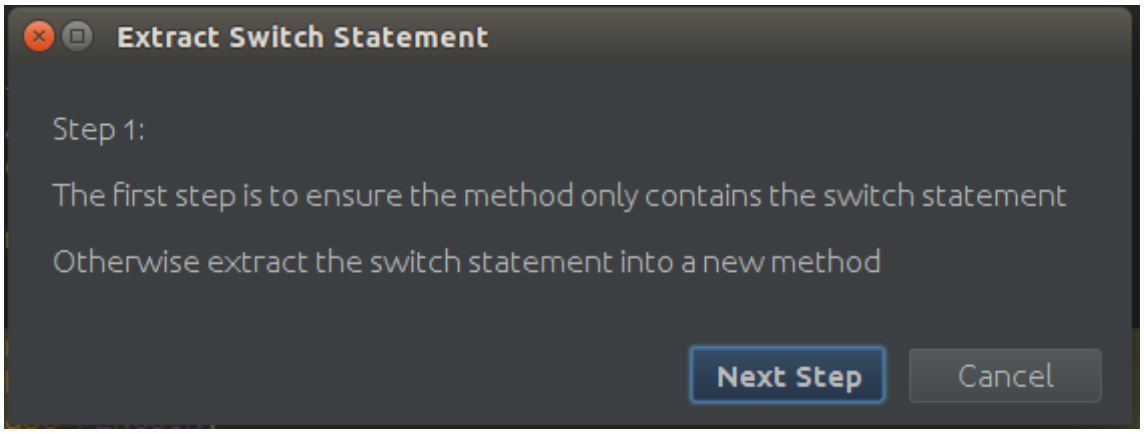

Figure 3.14: Replace conditional with polymorphism refactoring dialog

3.3 Software Development Process

3.3.1 Incremental Development

During the development of the IntelliJ plugin, we have employed the *Incremental Development* method, which is also referred to as the "Incremental build model" in Wikipedia [23]. Incremental evelopment slices the functionality of the software into smaller components, and each slice of the functionality is designed, implemented and delivered separately. The software is finished when all its functionalities are completed.

During the design stage of this project, we notice that the structure of the plugin we aim to build naturally fits well with the increment development approach. Each code smell targeted in the plugin can be classified as a functionality, and the feature in one code smell is clearly isolated from another. This makes it easy to develop the detection and correction of code smells incrementally

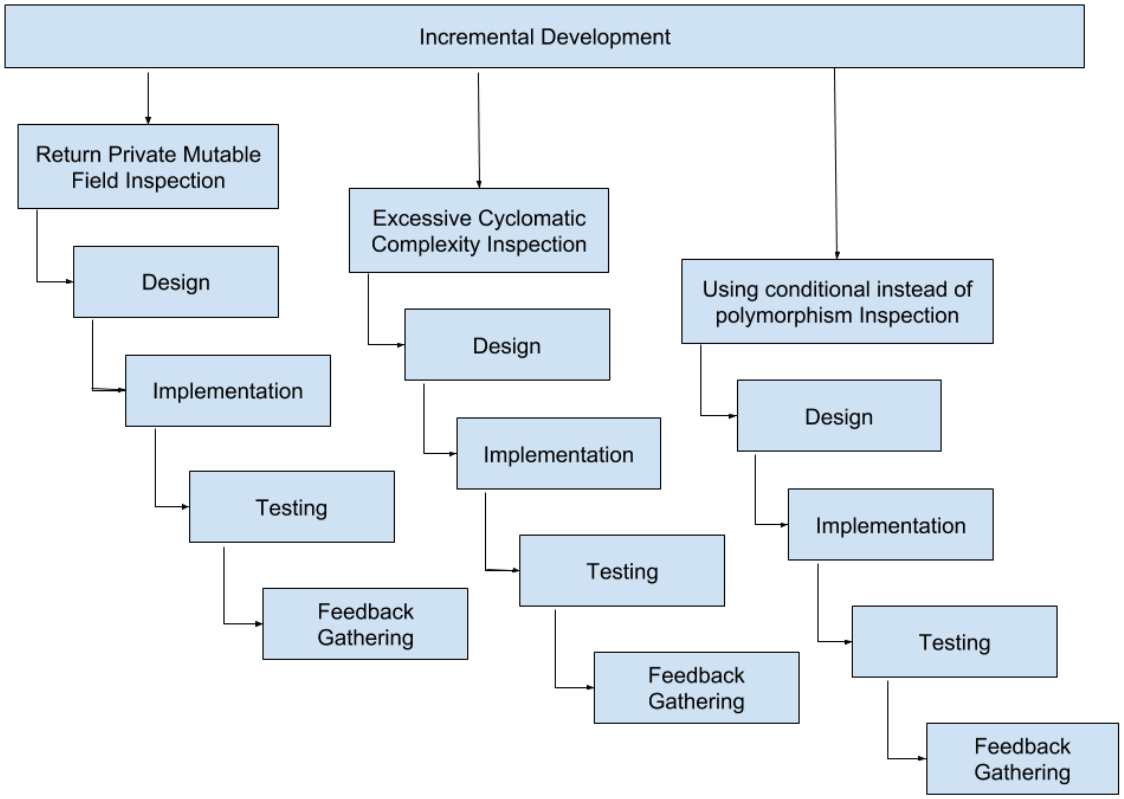

Figure 3.15: Incremental development of code inspections

one after another. Figure 3.15 shows the incremental development overview of the code inspections in this project.

There are many advantages we benefit from when employing the incremental development technique. Firstly, the initial version of the plugin was developed in a short period of time. Although the first version only includes the detection and correction of a single code smell, it greatly helps to decide the direction of the project, and find out the scope should be covered as part of the project.

Secondly, we gathered a lot of feedback from users at the early stage because the user testing can be done as soon as one functionality (the detection and correction of a code smell) is completed. We then use the gathered feedback to improve not only the current code smell detection but also the ones we will be implementing afterwards. This greatly shortens the feedback loop in the project, and will be one of the reasons towards the success of the project evaluation (see Chapter 7).

Thirdly, the testing and debugging become much easier throughout the development process. Since the code smell detection and corrections are developed one by one, so fewer changes are made at a time and they are generally only made to a specific type of code smell. For example, when there is a bug in the refactoring of the using conditional instead of polymorphism code smell, we can not only quickly identify the bug in the source code, but also efficiently eliminate the bug. This is because we can focus on a specific functionality of the plugin without paying much attention to unrelated parts of the system.

Although the incremental development approach helps to define the project scope quickly, shortens the feedback loop, and speeds up testing and debugging. One issue has been encountered when using the incremental development approach. As the functionalities of the plugin are developed incrementally, it is difficult to have an accurate estimation of the development time and effort required. We had to draft a project plan at the beginning, then continuously adjust the plan as we make more progress. Although we take advantages from the development aspect, we

eventually put a lot of effort to ensure the completion of the project and a successful delivery of all the functionalities.

3.3.2 Testing

In order to develop an IntelliJ plugin that is of a good quality, works as expected, and free of bugs, comprehensive testing has been conducted throughout the development process. As recommended in the IntelliJ Plugin DevGuide [24], we write model level tests on top of the test framework provided by the IntelliJ Platform. The model level functional tests focus on the behavior of a feature as a whole instead of smaller function units.

One of The biggest advantages using this approach is that the tests are run within an instance of IntelliJ IDEA, which ensures that the testing environment is as similar to the production environment as possible. Another benefit is that the tests are very stable. Since the tests focus on only the behavior of the plugin, so the tests are not affected by any changes to the underlying implementation, which makes the tests solid and easy to maintain.

Code Highlighting Testing

When a code smell is detected, the code block that directly contains the code smell should be highlighted by the plugin in order to warn developers. In the tests, we would like to test this behavior to ensure that the right piece of code is highlighted with the correct description of the code smell. The testing framework provided by IntelliJ Platform provides explicit support for testing different kinds of code highlighting. The Listing 3.7 below shows an example of the dedicated markup format for this task.

```
<warning descr="use conditional instead of polymorphism">
    switch (type) {
        case PENGUIN:
            return 1;
        case EUROPEAN:
            return 2 * baseSpeed;
        case NORWEGIAN_BLUE:
            int totalSpeed = baseSpeed > 10 ? baseSpeed : 10;
            return totalSpeed;
        default:
            return 0;
    }
</warning>
```

Listing 3.7: Dedicated XML markup for testing code highlighting

For each of the code smell inspections in the plugin, we create a set of test cases covering different cases might occur in real world applications. Each test case is annotated with this markup format, then we use the IntelliJ test runner to run the code inspections on the test cases and assert that code smell can be correctly identified and the smelly code is indeed highlighted.

Refactoring Testing

After testing the code can be highlighted correctly, we also need to ensure that refactoring can be done as expected. We use the following approach: At first, a code example that contains a code smell is prepared, we define it as the original version. Then we duplicate the original version

and manually fix the code smell using the same refactoring step as we designed in the plugin, so we produce a different version - a version without code smell. The IntelliJ test runner runs code inspections on the original version, applies refactoring steps on the code and finally compares the code with the version without code smell. As the test produces the desired result, we can ensure that the refactoring works as expected.

Swing UI Testing

In addition to the code inspection and refactoring features, there are many Swing UI components involved in the refactoring process. We have to make sure that they render the correct information and display at the target position, so UI testing is also an important element. However, this is missing from the test framework provided by IntelliJ Platform, there is no support for testing Swing UI components. Although there are testing libraries such as Sikuli [25] which provide different kinds of GUI testing, we believe that the time and effort spent on UI testing outweighs the benefit gained. So we decide not to use them and conduct manual testing instead.

3.3.3 Documentation

As the plugin is published to the JetBrains Plugins Repository, we would like to ensure any user of this plugin receives the support they need. Although extra care has been taken to ensure that the user interface is easy to understand and steps can be followed intuitively, for users who experience difficulty using the plugin, detailed download and installation instructions and common FAQs are documented in the README file.

On the other hand, comments are rarely used throughout the source code of the project. This is mainly to adhere to the "good code should be self-documenting" principle outlined in Robert Martin's book [26].

Chapter 4

Replace Conditional With Polymorphism Inspection

One of the code smells tackled in this project is so called "using conditional instead of polymorphism", as discussed in Section 2.2.1. The detection and correction of this code smell is codified into the plugin as the *Replace Conditional With Polymorphism Inspection*.

This chapter discusses the approaches used to identify the code smell, refactoring methods applied to eliminate the code smell, as well as any limitations associated with the chosen techniques.

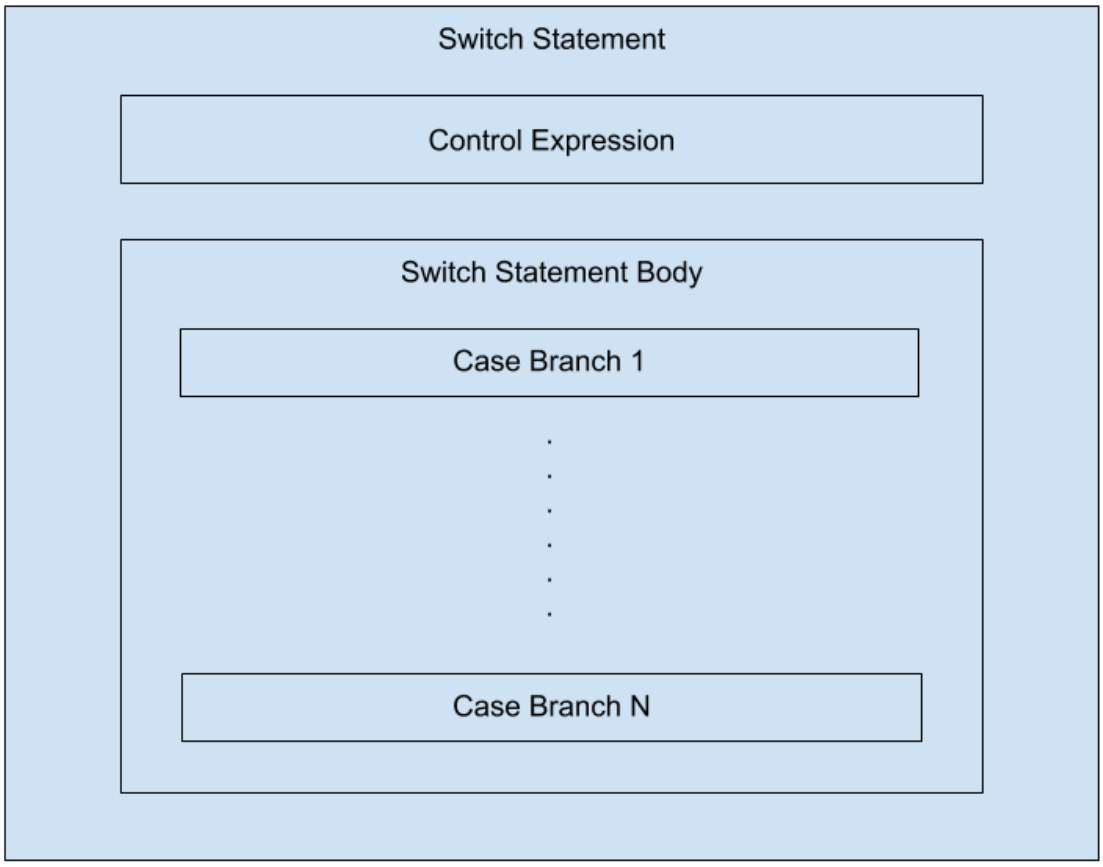

Figure 4.1: Switch statement structure overview

4.1 Identification

We propose the following approach to identify the using conditional instead of polymorphism code smell:

1. Find all the switch statements in the file currently edited by the developer.
2. Examine each of the switch statements, and determine whether there is a potential code smell based on its structure (see Figure 4.1).

Switch Statement Visitor

In order to find all the switch statements from the current file, the *visitor pattern* is used. We build a *JavaElementVisitor* to traverse the PSI tree in the file and observe every switch statements visited. By overwriting the *buildVisitor()* method, the visitor is run against the PSI tree on every code changes made and in turn re-triggers the switch statement analysis.

```
@NotNull
@Override
public PsiElementVisitor buildVisitor(final ProblemsHolder holder,
                                      boolean isOnTheFly) {
    return new JavaElementVisitor() {
        @Override
        public void visitSwitchStatement(PsiSwitchStatement statement) {
            //......
        }
    };
}
```

Listing 4.1: Overwrite buildVisitor() with switch statement visitor

Examine Switch Statement

When examine the structure of a switch statement, there are two pieces of information that are of interest in the code smell detection, and they are used together to determine if the switch statement contains the code smell:

- the number of case branches in the switch statement body - we define a branch threshold and the switch statement is classified as containing code smell only if it contains a number of case branches exceed the threshold. The threshold is defined mainly for two reasons: 1) this provides flexibility to the plugin users and it is up to them to decide what they think is a good threshold for the branch counts. 2) the default threshold is set to 2 as a switch statement with only two branches could simply be converted to an if statement as shown in Listing 4.2 and Listing 4.3.
- the control expression of the switch statement - we check that the control expression is either a reference to a field (which represents a property of a class) or a method call. This is the second condition that the switch statement has to satisfy in order to be classified as containing code smell.

```
switch (specialNumber) {
    case 1:
        return 10;
    default:
        return -1;
}
```

Listing 4.2: Switch statement with only two branches should not be classified as code smell

```
if (specialNumber == 1) {
    return 10;
} else {
    return -1;
}
```

Listing 4.3: Example of converting switch statement to if statement

Figure 4.2 below illustrates the procedures followed to determine whether a switch statement contains a code smell.

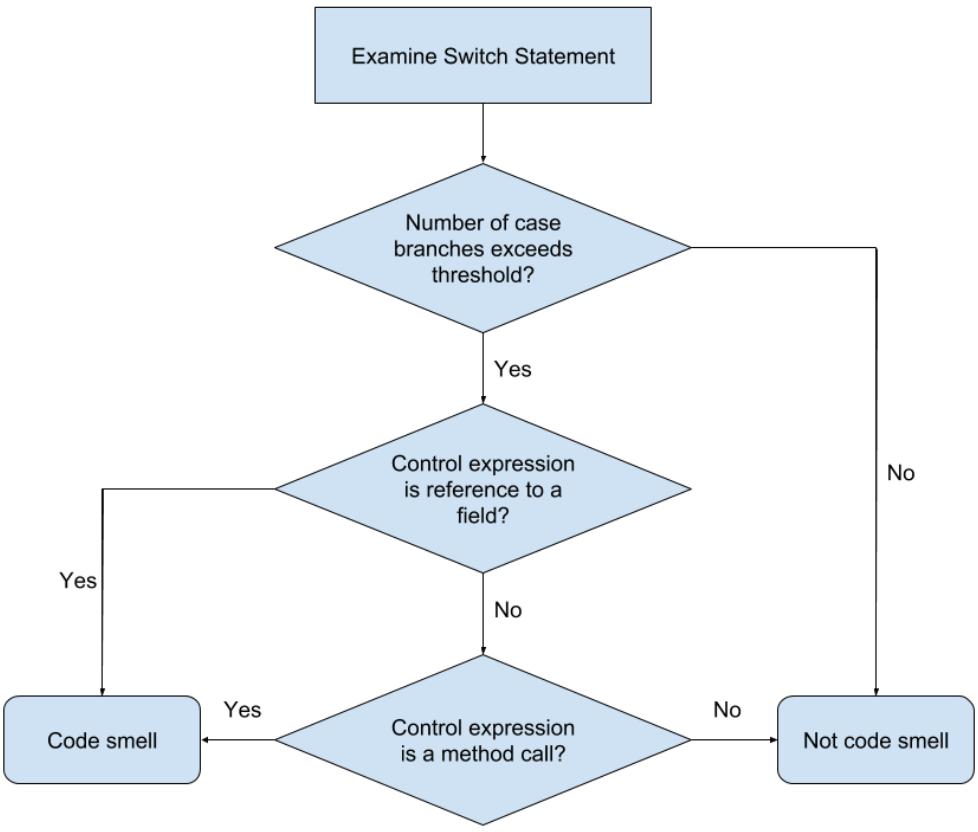

Figure 4.2: Procedure to examine switch statement

4.2 Refactoring

In the following sections, we first present an example walk-through of the step by step refactoring process and follow by a more detailed discussion on each refactoring step.

4.2.1 Example Step-By-Step Walkthrough

To give an overview of the step by step refactoring process, we use a flowchart as shown in Figure 4.3 below to illustrate the refactoring steps involved to transform a switch statement with code smell to an object-oriented design using polymorphism.

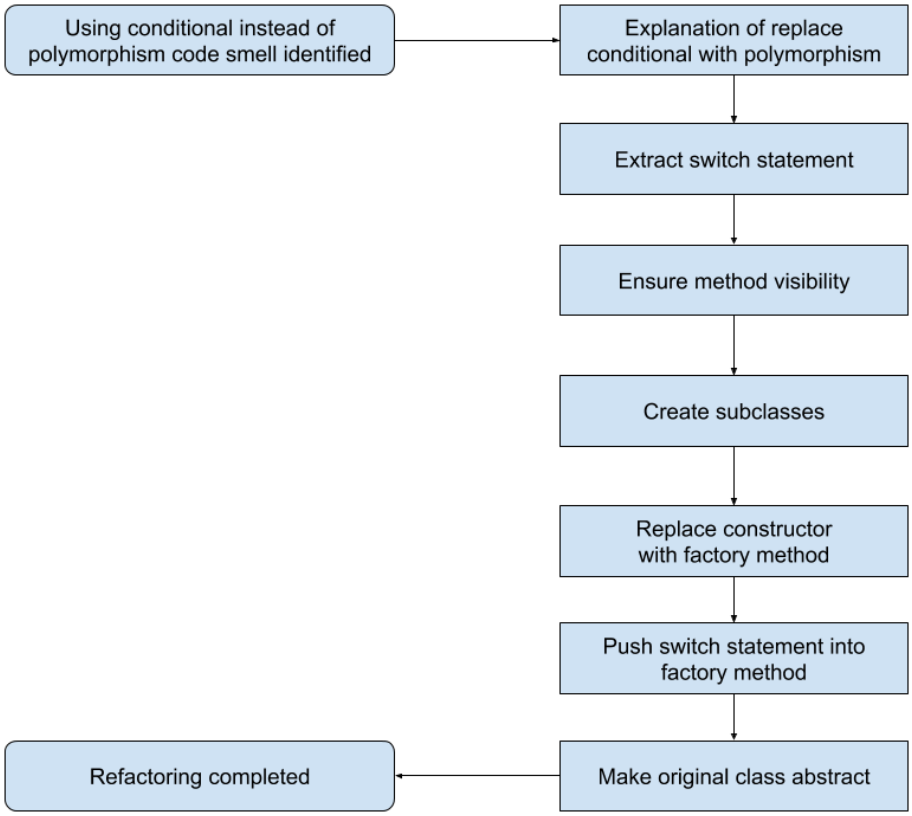

Figure 4.3: Refactoring steps overview

Although it is clear to see each of the refactoring step involved, it might not be obvious to see that how and why each of the refactoring step transforms the smelly code towards the desired state. We will present a concrete code smell example below in Listing 4.4, go through the refactoring process step by step, and demonstrate how the code transformations occurred throughout the steps.

Explanation of replace conditional with polymorphism

The refactoring process does not start with a refactoring action, but instead with an explanation of the code smell. At this step, no modifications are made to the source code. This step acts as the entry point of the refactoring process, it allows the developer to gain a deeper insight into the code smell and decide if they would like to start the actual refactoring steps. More details about the explanation will be further discussed in Section 5.2.2 below.

```java
public class Bird {

    enum BirdType {
        PENGUIN,
        SPARROW,
        PARROT
    }

    private BirdType type;
    private int baseSpeed;

    Bird(BirdType type, int baseSpeed) {
        this.type = type;
        this.baseSpeed = baseSpeed;
    }

    public int getSpeed() {
        if (baseSpeed < 0) {
            return 0;
        }
        switch (type) {
            case PENGUIN:
                return 1;
            case SPARROW:
                return 2 * baseSpeed;
            case PARROT:
                int totalSpeed = baseSpeed > 10 ? baseSpeed : 10;
                return totalSpeed;
            default:
                return 0;
        }
    }

    public static void main(String[] args) {
        Bird aBird = new Bird(BirdType.PENGUIN, 1);
        System.out.println(aBird.getSpeed());
    }

}
```

Listing 4.4: Code smell example before refactoring

Extract switch statement

This is the first step that a refactoring action might be executed to make changes to the source code. There are two phases in this step. The first phase is to observe the structure of the method that contains the switch statement. The second phase is only conducted if the method contains not only the switch statement but also other statements. In Listing 4.4, the **getSpeed()** method contains the target switch statement as well as an if statement. Phase two is then to extract the switch statement into a separate method. The switch statement is extracted into the new method **getBirdSpeed()** as shown in Listing 4.5.

```
public int getSpeed() {
    if (baseSpeed < 0) {
        return 0;
    }
    return getBirdSpeed();
}

private int getBirdSpeed() {
    switch (type) {
        case PENGUIN:
            return 1;
        case SPARROW:
            return 2 * baseSpeed;
        case PARROT:
            int totalSpeed = baseSpeed > 10 ? baseSpeed : 10;
            return totalSpeed;
        default:
            return 0;
    }
}
```

Listing 4.5: Switch statement extracted into separate method

Ensure method visibility

After the switch statement being extracted into a new method, we need to ensure that the method is not only visible with in the class, but also visible in the package level as this method will be override in subclasses in the future steps. Listing 4.6 shows the changes in the modifier of the method.

```
//before
private int getBirdSpeed() { ... }

//after
protected int getBirdSpeed() { ... }
```

Listing 4.6: Extracted method visibility before and after the refactoring step

Create subclasses

The next step is to create subclasses for each of the case branches in the switch statement. Inside each subclass, the constructors are created to match with the super class, and the method created following the previous step is also overridden. Listing 4.7 shows the overview of the four subclasses created, each of them extends the class **Bird** and overrides the **getBirdSpeed()** method with the behavior in the corresponding case branch in the switch statement. Only the implementation of the class **Penguin** is shown, the implementation of other classes are omitted as they follow the same pattern.

```
public class Penguin extends Bird {
    Penguin(BirdType type, int baseSpeed) {
        super(type, baseSpeed);
    }

    @Override protected int getBirdSpeed() {
        return 1;
    }
}

public class Sparrow extends Bird { ... }

public class Parrot extends Bird { ... }

public class DefaultBird extends Bird { ... }
```

Listing 4.7: Create subclass for each of the case branch

Replace constructor with factory method

After the subclasses are created, the refactoring process follows by replacing the constructors in **Bird** class with factory methods. As shown in Listing 4.8, a number of changes are made to the source code:

1. The static factory method **createBird()** is created

2. The direct instantiation of the **Bird** class is adapted to use the factory method

3. Also to note that, the visibility of the constructor is downgraded to package level so that the use of the factory method is enforced in order to instantiate an object.

```
Bird(BirdType type, int baseSpeed) {
    this.type = type;
    this.baseSpeed = baseSpeed;
}

public static Bird createBird(BirdType type, int baseSpeed) {
    return new Bird(type, baseSpeed);
}


//before
Bird aBird = new Bird(BirdType.PENGUIN, 1);

//after
Bird aBird = createBird(BirdType.PENGUIN, 1);
```

Listing 4.8: Replace constructor with factory method

Push switch statement into factory method

The next step in the refactoring process is to move the switch statement into the factory method created in the previous step, and instantiate the corresponding subclass in each of the case branch. We can see from Listing 4.9 below, the correct subclass would be created based on the enum **BirdType** in the factory method. For example, when the **BirdType** is PENGUIN, a **PenguiBird** is instantiated.

```
static Bird createBird(BirdType type, int baseSpeed) {
    switch (type) {
        case PENGUIN:
            return new Penguin(type, baseSpeed);
        case SPARROW:
            return new Sparrow(type, baseSpeed);
        case PARROT:
            return new Parrot(type, baseSpeed);
        default:
            return new DefaultBird(type, baseSpeed);
    }
}
```

Listing 4.9: Create the correct object in factory method

Make original class abstract

The last step in the refactoring process is to make the original class abstract to prevent it from being instantiated. The original switch statement is now deleted as it is not useful any more, the method that contains the switch statement is also made abstract as it is overridden in the subclasses. The above changes are shown in Listing 4.10.

```
public abstract class Bird {

    //...

    protected abstract int getBirdSpeed();

    //...

}
```

Listing 4.10: The last step in the refactoring process

The refactoring process includes a total of seven steps to transform a switch statement towards a more object-oriented design through the use of polymorphism. Each of the steps will be explained in more details and any edge cases being handled are also discussed in the below sections.

4.2.2 Explanation of replace conditional with polymorphism

The intentions of the IntelliJ plugin are not only assisting in the identification and refactoring process of code smells, but also aiming to help developers to learn about different code smells. This makes the explanation of the code smell essential and we believe that explaining the code

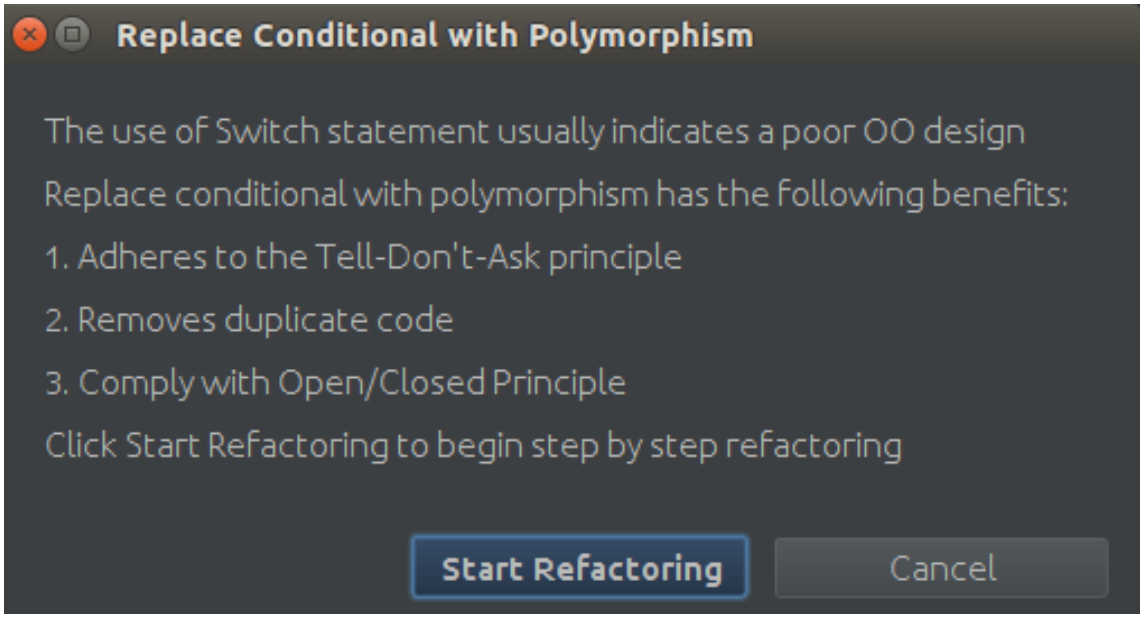

Figure 4.4: Explanation of replace conditional with polymorphism

smell to the developers before they dive into the refactoring process directly is significant in order for them to understand why the refactoring is needed and what are the benefits gained after the refactoring.

The explanation is provided through the use of a dialog as shown in Figure 4.4.

In addition, as the entry point of the refactoring process, this step also allows the developers to cancel the refactoring process if they believe the code smell is not relevant after reading the explanations.

4.2.3 Extract switch statement

The first step in the refactoring process is to ensure the target switch statement is the only statement in the method to which it belongs. There are two phases in this step: 1) Observe the structure of the method that contains the switch statement; 2) Perform the extract method refactoring action to extract the switch statement into a separate method.

In order to provide the users with enough information about the refactoring step so that they fully understand the upcoming refactoring action, the dialog in Figure 4.5 is displayed.

After the user clicks the "Next Step" button, the phase one of this step is started. The observation of the method structure has a single goal - determine if the method contains any other statements apart from the switch statement. This is done by making use of the *getStatements()* API on the method body as shown in Listing 4.11 below.

```
if (method.getBody().getStatements().length > 1) {
    //perform action
}
```

Listing 4.11: Perform action if method contains more than one statements

It is important to make sure that the method contains only the target switch statement. This is because this method would be overridden by subclasses created in later steps, any other statements

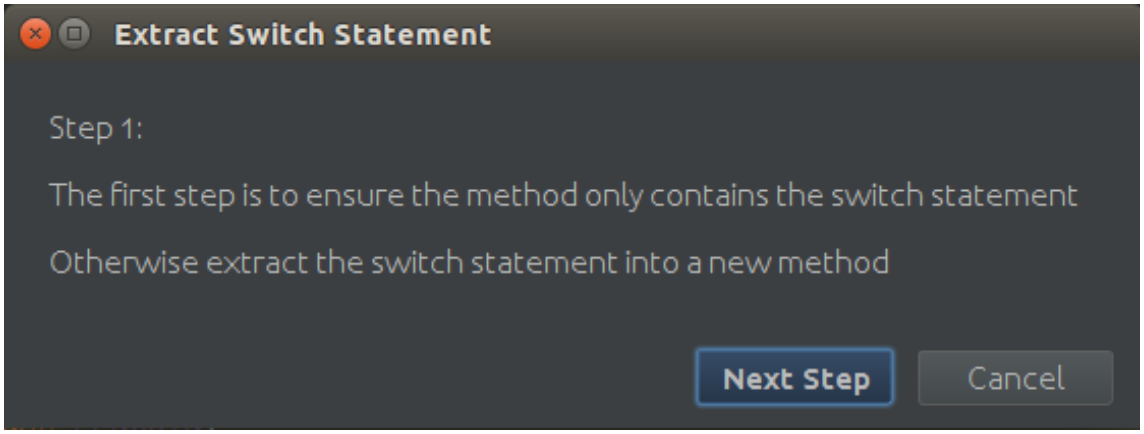

Figure 4.5: Extract switch statement dialog

in the method would have to be duplicated so that the existing logic is maintained. In oder to simplify the refactoring steps, we perform the extract method action to ensure the method contains only the target switch statement.

Instead of implementing the extract method action, we make use of the existing extract method action provided by IntelliJ Platform. This ensures a consistent user interface and behavior as this is the same action being used when a developer tries to extract a method in IntelliJ IDEA. We also take the advantages that the built-in refactoring action are able to handle a lot of edges cases real world applications might have. Although the built-in utility is convenient to use, it has limitation on its flexibility. For example, we would like to ensure the extracted method is at least package visible but the built-in action allows the user to set the visibility of the method to private by default. Hence, we introduce the next step in the refactoring process in Section 4.2.4 below.

4.2.4 Ensure method visibility

The extracted method from the first step will be overridden in subclasses in later steps. In order for a method to be overridden by subclasses, it has to be accessible from the subclasses, which means its visibility should not be private. As discussed above, the use of the IntelliJ built-in extract method action does not enforce the visibility of the extracted method, so the method visibility is guaranteed during this step.

On start of the step, the below dialog in Figure 4.6 is displayed to present a description of the current step to the users.

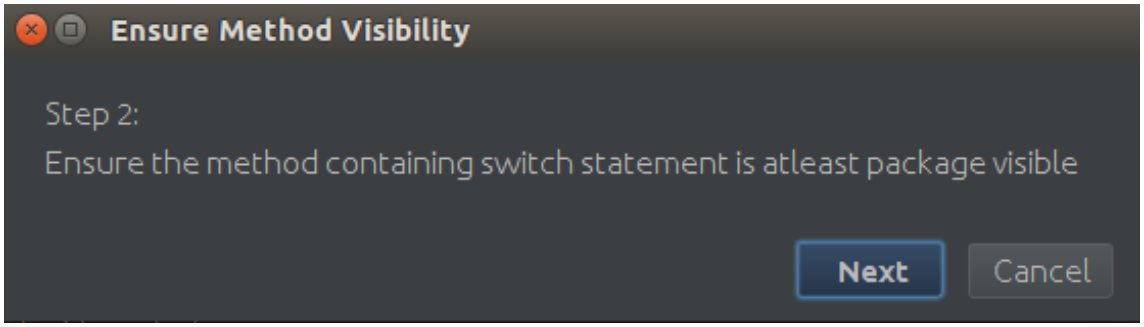

Figure 4.6: Ensure method visibility dialog

As shown previously in the step by step walk-through, the access modifier of the method is modified in Figure 4.6. To note that, the visibility of the method is only updated if it is in private level, otherwise the visibility of the method is unmodified to preserve the original intention.

4.2.5 Create subclasses

The next step in the refactoring process is to create subclasses for each of the case branch in the switch statement. It is obvious to see that the first two steps are prerequisites to this step and they ensure that the source code is in the desired structure before the refactoring actions take place during this step.

As usual, a dialog is displayed and awaits the user's confirmation before starting the refactoring as shown in Figure 4.7.

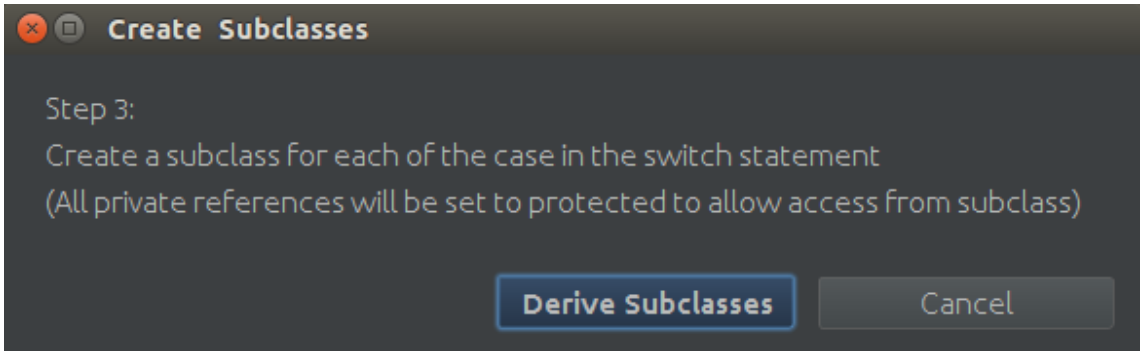

Figure 4.7: Create subclasses dialog

Unlike the previous step where the IntelliJ built-in extract method functionality is used, we decide to implement our own refactoring action during this step. This is mainly because the refactoring action performed in this step is specific to the need of this plugin and the built-in functionalities provided by IntelliJ do not fit for this purpose. We not only create subclasses to extend the superclass, but also create constructors to match the superclass and override the method that contains the switch statement. It is impossible to find a built-in functionality in IntelliJ Platform that does all the required operations. In addition, the action we implement provides greatly flexibility and it could adapt to any future changes easily.

The subclasses are created automatically with all the necessary methods filled in after the user clicks "Derive Subclasses". However, there are multiple steps involved to perform all the operations required. The list of steps involved are listed below in the order they are executed:

1. The first step is to identify the case branches from the switch statement
2. For each of the case branches, we extract some key informations that is useful when creating deriving the subclasses
3. Create subclass for each of the case branches
4. Duplicate the method that contains the switch statement in the subclass
5. Annotate the method with the *Override* annotation to indicate it overrides method from superclass
6. Replace the method body with any code that is inside the corresponding case branch
7. Add constructors to the subclass to match with the superclass, so that the subclass can be constructed

The Figure 4.12 below illustrates the order of the above steps occurred in the creation of a subclass.

```
public class Penguin extends Bird {                    //step 1
    Penguin(BirdType type, int baseSpeed) {            //step 5
        super(type, baseSpeed);
    }

    @Override                                          //step 3
    protected int getBaseSpeed() {                     //step 2
        return 1;                                      //step 4
    }
}
```

Listing 4.12: Subclass example with creation steps labeled

Although the above steps might seem to complete the whole process of creating subclasses from case branches, there is one extra step that was not mentioned but essential to ensure the code still compiles after the refactoring. During step 4, any code inside the case branch is moved into the subclass, that means the scope of the code has been changed. It is important to ensure that any references the code can access originally are still accessible from the subclass. In order to address this subtle issue, the visibility of every references being used in the code block are examined and adjusted. The Listing 4.13 below shows the changes made to the modifier of the **baseSpeed** field to allow access from the **Sparrow** subclass.

```
public abstract class Bird {
    //before
    private int baseSpeed;

    //after
    protected int baseSpeed;


    //...
}


public class Sparrow extends Bird {
    Sparrow(BirdType type, int baseSpeed) {
        super(type, baseSpeed);
    }

    @Override protected int getBaseSpeed() {
        return 2 * baseSpeed;
    }
}
```

Listing 4.13: Changes to the private field modifier

Subclass Naming

When creating the subclass, a lot of effort has been put to ensure a meaningful name is given to the subclass based on the information gathered from the switch statement. Since each of the case branch has a different case value (**BirdType** in the example above), we believe that the case value is a good representation of the subclass and can distinguish them from each other. A number of approaches have been tried to generate the subclass name from the case value, the list of subclass

names generated using different approaches are shown below:

1. BirdPENGUIN
2. PENGUINBird
3. PenguinBird
4. Penguin

Although all above generated names are valid class names in Java, the first two do not comply with the general conventions and left with the last two options. Both the last two approaches convert the case value to camel case as this is the commonly accepted convention used for Java class names. The difference is that the first one includes the base class name while the latter does not. We present both options to real users and their feedbacks are gathered. The feedback from users show that the last option tends to be a more natural class name. The exception is that when naming the subclass created corresponding to the default case in the switch statement, the third approach is used since it would be more meaningful to name the class "DefaultBird" instead of "Default".

It is also worth to point out that, these automatic generated class names are served only as a suggestion to the developers. If a different name sees a better fit, the names can be easily changed by the developers manually using IntelliJ built-in "Rename" functionality.

4.2.6 Replace constructor with factory method

The next step in the refactoring process is to replace the constructors with factory methods so that the correct subclass is instantiated.

When the factory method is created, a default name is provided as shown in Figure 4.8. This is useful for inexperienced developers as they might not be able to give the factory method a meaningful name at this stage, the automatic generated name can be a temporary placeholder and might be used as a hint for them to produce a more appropriate method name in a later stage. For developers who are more familiar with the context and more confident in naming the factory method, the dialog also provides the flexibility for them to change the method name directly.

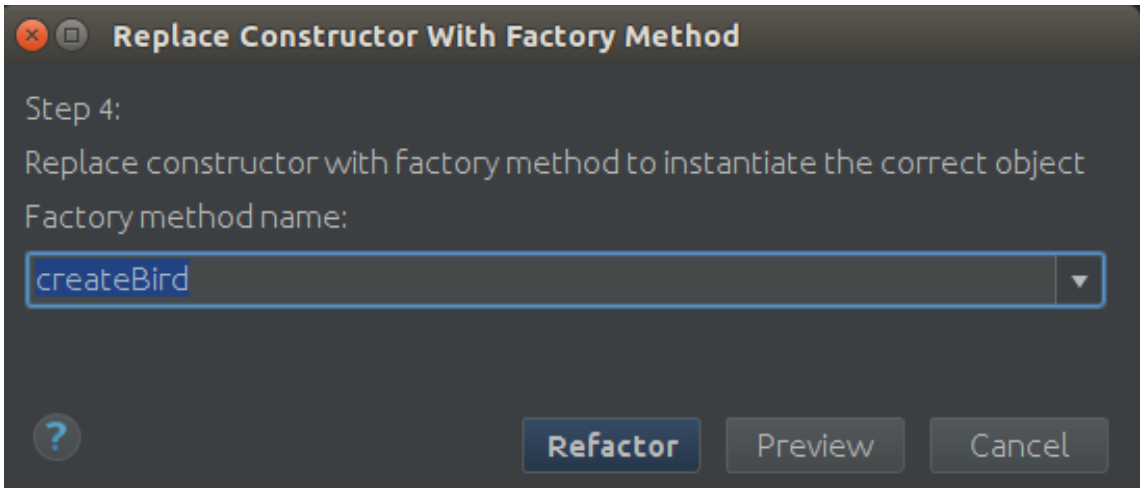

Figure 4.8: Replace constructors with factory method dialog

During this step, for each of the constructors in the superclass, we make use of the IntelliJ built-in "Replace constructor with factory" refactoring action to perform the transformation as shown earlier in Listing 4.8. When a constructor is replaced with a factory method, it also has

a side effect - the use of the original constructor across the project are replaced with the call to the new factory method. This step is crucial because it ensures that the developers do not have to manually modify the code to adapt to the any changes made by the plugin and the source code still compiles after this step. These kind of considerations are taken into account during the development of the plugin to ensure users go through the refactoring process smoothly.

4.2.7 Push switch statement into factory method

Following the factory methods are created to replace the original constructors, this step is to push the switch statement into the factory methods so that the right object is selected to be instantiated as described in the dialog below.

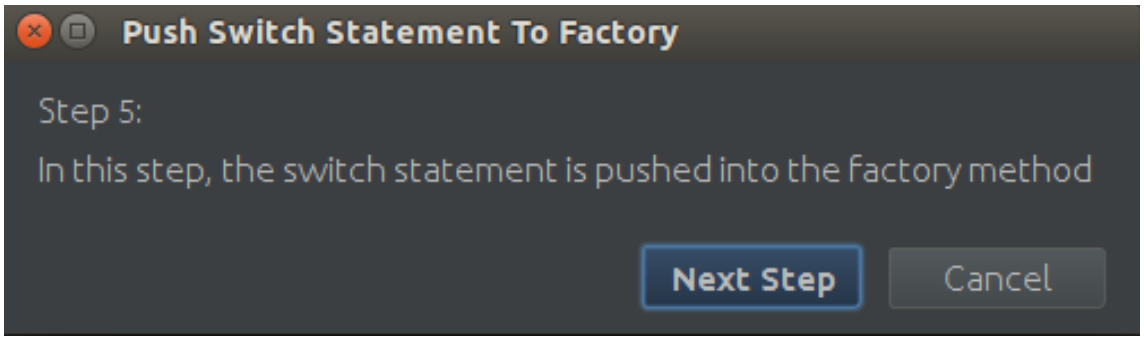

Figure 4.9: Push switch statement into factory method dialog

During the previous step, we make use of the built-in IntelliJ functionality to create the factory methods. Although this is convenient to use, it introduces a number of problems we have to work around at this step. One of the problems we experienced is that we do not have the reference to the created factory methods after the "Replace constructor with factory" action in IntelliJ is triggered. The name of the factory methods are also unknown at this point, as the developers could change the factory method name during the last step. We propose an approach to work around this issue as illustrated below in Figure 4.10.

This approach is used due to the limitation of the IntelliJ built-in refactoring action. Although this approach can determine the factory methods reliably, we are unable to produce a more elegant solution. It is hoped that a potential callback mechanism could be introduced into the APIs so that any newly created methods are to remove the necessity of these kind of workarounds in the plugins. After the factory methods are identified, the switch statement is then copied and moved into each of the factory methods. This refactoring step is finished by going through the switch statement and replacing the code inside each case branch with the instantiation of the correct subclass. The final outcome of this step is shown in Listing 4.14.

```
static Bird createBird(BirdType type, int baseSpeed) {
    switch (type) {
        case PENGUIN:
            return new Penguin(type, baseSpeed);
        case SPARROW:
            return new Sparrow(type, baseSpeed);
        case PARROT:
            return new Parrot(type, baseSpeed);
        default:
            return new DefaultBird(type, baseSpeed);
    }
}
```

Listing 4.14: Correct subclass is instantiated in the factory method

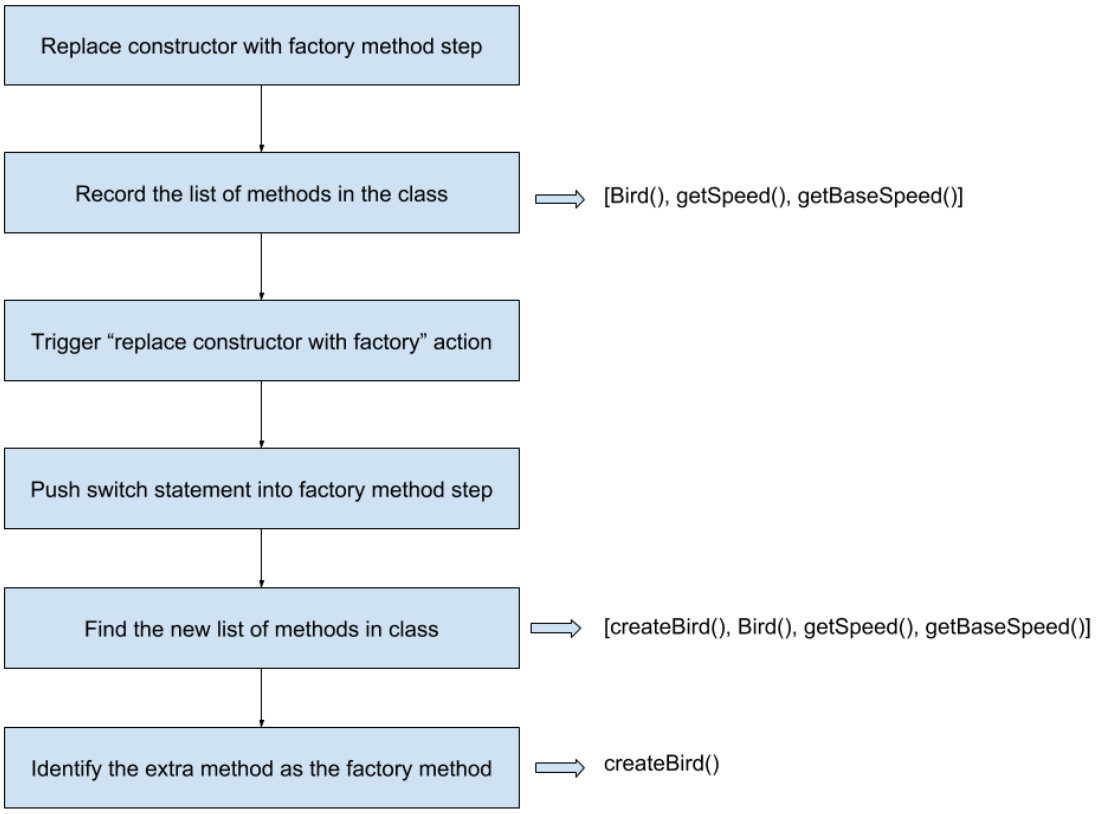

Figure 4.10: Work around to find factory method

4.2.8 Make original class abstract

The last step in the refactoring process is to make the original method that contains the switch statement abstract and in turn declare the class abstract as outlined in the dialog below. Declaring the class abstract would prevent the class from being instantiated, and it enforces one of the subclasses to be used instead.

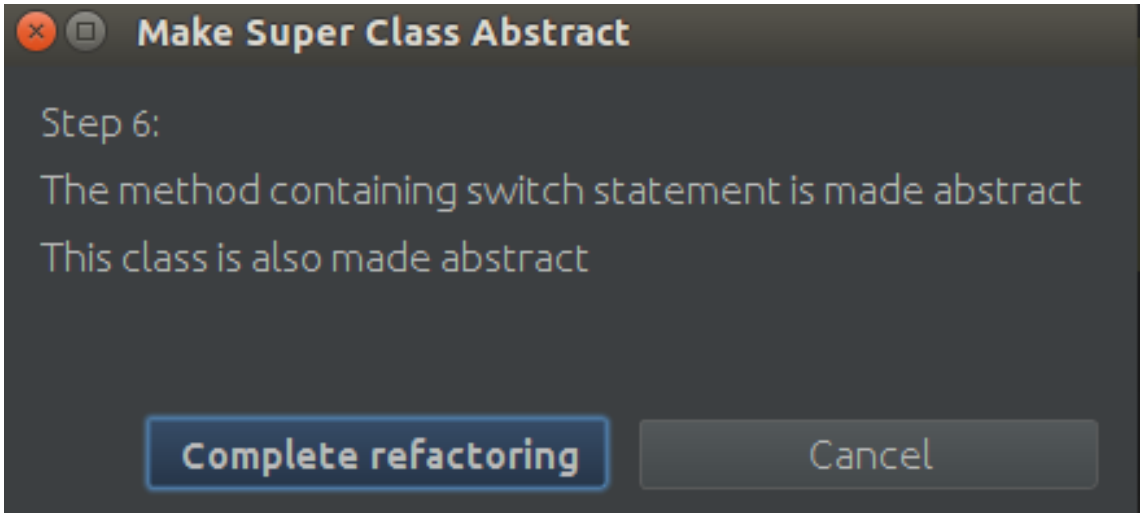

Figure 4.11: Last step in the refactoring process

In the previous steps, the visibility level of the methods or class fields are updated through the modification to their list of modifiers. The same approach is used to make the class abstract during

this step. When adding the abstract modifier to the method, the method body is also removed as it is no longer useful as shown in Figure 4.10.

4.3 Limitation

Although the replace conditional with polymorphism inspection is able to achieve a high degree of accuracy in real world applications for both detection and correction (see Chapter 7), there are edge cases not being covered and limitations associated with the existing techniques and methods. In the sections below, we will realize any limitations in detection and correction and discuss potential improvements to the current approach.

4.3.1 Detection Limitation

The detection phase of the replace conditional with polymorphism code inspection mainly focuses on the structure of the switch statement. Although this strategy provides a reliable identification of the using conditional with polymorphism code smell, it has weaknesses that might introduce false code smell identification.

Firstly, the default setting of the branch count threshold is static and therefore might not be optimal to the project a developer is working on. Configuring the threshold value requires the developer to have a certain degree of understanding of the code smell, a developer with less experience might set a threshold that is too low or too high and eventually result in a poor identification of the code smell. One of the improvements might be to calculate the default threshold based on an initial analysis of the whole code base. but that is out of the scope of this project.

Secondly, the control expression of the switch statement is analyzed to determine if the code smell exists. One of the considerations for the control expression is that the evaluated value of the control expression should stay unchanged through the lifetime of the object, this ensures that the behavior of the object which is based on the control expression is also unchanged. However, we are not able to propose a reliable technique to validate this. We instead provide the flexibility to the user and it is up to the user to decide if there is an underlying problem in the detected code smell.

4.3.2 Correction Limitation

During the correction phase of the code inspection, our approach of always ensuring the code is structured in the desired shape before proceeding to the next step greatly mitigates the risks of breaking the existing code during the refactoring process and handles most of the edge cases.

However, one of the edge cases the current refactoring technique does not deal with very well is when the class has multiple constructors. As one of the steps during refactoring pushes the switch statement into each of the constructors, even though it improves the code through the use of polymorphism, it introduces a potential code duplication problem. In the future versions of the plugin, a different refactoring strategy might be implemented targeting specifically this case.

Chapter 5

Excessive Cyclomatic Complexity Inspection

The second code smell tackled in this project is so called "excessive cyclomatic complexity", as discussed in Section 2.2.2. The detection and correction of this code smell is codified into the plugin as the *Excessive Cyclomatic Complexity Inspection*.

This chapter discusses the approaches used to identify the code smell, refactoring methods applied to eliminate the code smell, as well as any limitations associated with the chosen techniques.

5.1 Identification

Although cyclomatic complexity can be used to indicate complexity of programs, classes, methods and even statements, we focus on the identification of this code smell in the method level in this project. We propose the following strategy to identify methods with excessive cyclomatic complexity:

1. Find all the methods in the file currently edited by the developer
2. Examine each of the methods, calculate its cyclomatic complexity and determine whether the method suffers from the excessive cyclomatic complexity code smell based on the predefined threshold.

Cyclomatic Complexity Threshold

The cyclomatic complexity threshold is one of the most important elements in the detection of the excessive cyclomatic complexity code smell. Although the cyclomatic complexity metric could be used to evaluate a method consistently and generate its complexity. A method with cyclomatic complexity 8, for example, might seem too complex for one developer but instead might be totally fine for a different developer. It is important to understand what does it mean for a method to have "excessive" cyclomatic complexity, and the threshold is introduced to serve this purpose.

After the cyclomatic complexity of a method is calculated, the complexity is compared against the predefined threshold. If the complexity is greater than the threshold, the method is classified as suffering the excessive cyclomatic complexity code smell.

For developers who are not familiar with the cyclomatic complexity, they might not be able to define a good threshold. Hence a default threshold 10 is defined to help developers to get started

without worrying about setting a threshold. The choice of the default threshold 10 is discussed in more details in Section 2.2.2.

Although the default threshold eases the processing of finding an initial threshold, its performance varies on different projects. Once a developer has a better understanding of the cyclomatic complexity, he might decide to set a threshold value that would result in a better performance in the project he works on. The Figure 5.1 below illustrates where a developer changes the default cyclomatic complexity threshold to 8. With a lower threshold, the detection would be more sensitive and more methods would be classified as containing code smell.

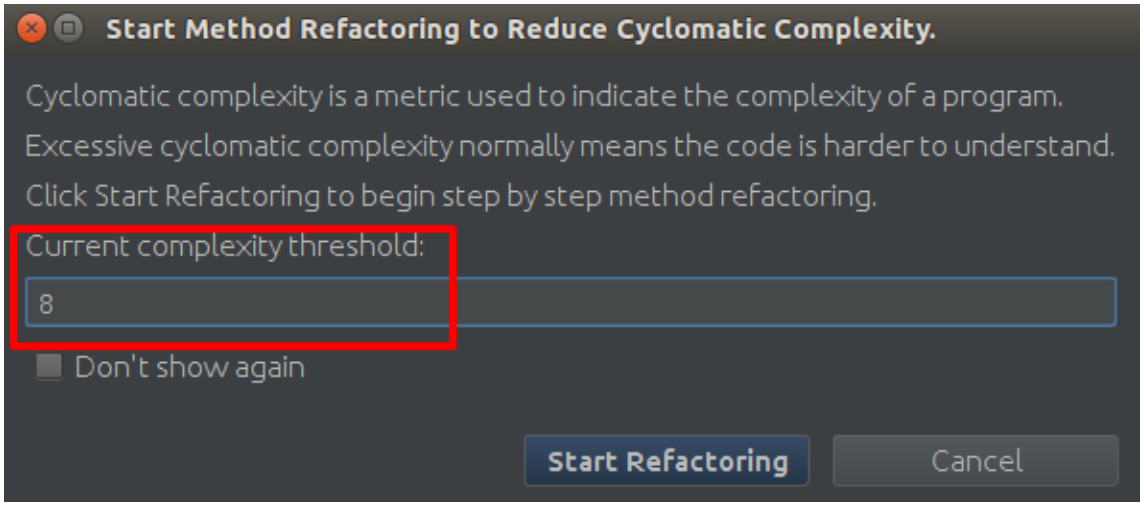

Figure 5.1: Dialog allows adjustments to complexity threshold

Cyclomatic Complexity Calculation

Cyclomatic complexity is calculated by counting the number of control flow statements in the method. The list of control flow statements includes if statement and for loop etc. Initially, we implemented a visitor on top of the *JavaRecursiveElementWalkingVisitor* to calculate the cyclomatic complexity. The visitor maintains a counter to keep track of the cyclomatic complexity. It then traverses the children elements in a method, when a control flow element is visited, it increments the cyclomatic complexity counter. The counter is read after each and every children elements in the target method are visited, and the counter represents the cyclomatic complexity of the method.

After this traversal algorithm is implemented, we found an existing *CyclomaticComplexityVisitor* which operates exactly the same strategy while calculating the cyclomatic complexity. In order to gain benefit from existing utilities, we decide to use the existing *CyclomaticComplexityVisitor* as it is well tested and provides a richer API.

5.2 Refactoring

In the following sections, we first present an example walk-through of the step by step refactoring process and follow by a more detailed discussion on each refactoring step.

5.2.1 Example Step-By-Step Walkthrough

To give an overview of the step by step refactoring process, we use a flowchart as shown in Figure 5.2 below to illustrate the refactoring steps involved to simplify a method with excessive cyclomatic complexity.

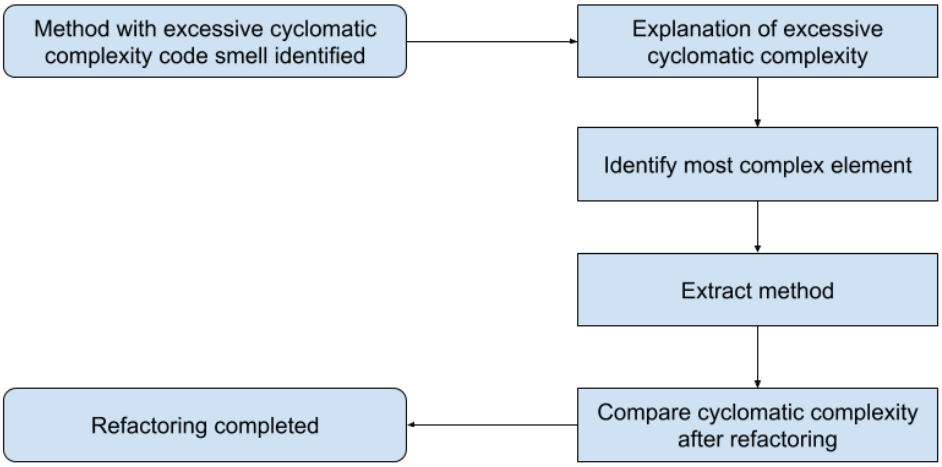

Figure 5.2: Refactoring steps overview

Although it is clear to see each of the refactoring step involved, it might not be obvious to see the operation performed by each of the refactoring step. We will present a concrete code smell example below in Listing 5.1, go through the refactoring process step by step, and demonstrate the underlying actions occurred throughout the steps. The method below suffers from the excessive cyclomatic complexity code smell and it currently has a cyclomatic complexity of 8. At the first glance, it is difficult to understand what this method does. After the refactoring, the method would be much easier to understand.

```
public void complexMethod() {
    if (date.after(new Date()) || date.before(lastYear)) {
        if (date.before(tomorrow)) {
            System.out.println(date.getTime() - tomorrow.getTime());
        } else if (date.before(startDate)) {
            System.out.println(startDate);
        }

        if (date.equals(s) || s.contains(date.toString())) {
            if (date.compareTo(yesterday) > 1) {
                return;
            }
            System.out.println(lastYear.getTime());
        }
    } else {
        System.out.println(date.getTime());
    }
}
```

Listing 5.1: Code smell example before refactoring

Explanation of excessive cyclomatic complexity

The refactoring process does not start directly with a refactoring action, but instead with an explanation of the code smell. At this step, no modifications are made to the source code. This step acts as the entry point of the refactoring process, it allows the developer to gain a deeper insight into the code smell and decide if they would like to start the actual refactoring steps. More

details about the explanation will be further discussed in Section 5.2.2 below.

Identify most complex element

The next step is to identify one of the children elements to be the most complex one. This is done by recursively analyze the cyclomatic complexity for each of the children elements, keep track of their cyclomatic complexities and the element with highest cyclomatic complexity is defined to be the most complex element. In the Figure 5.3 below, the most complex children element is selected with the rectangle.

```
public void complexMethod() {
    if (date.after(new Date()) || date.before(lastYear)) {
        if (date.before(tomorrow)) {
            System.out.println(date.getTime() - tomorrow.getTime());
        } else if (date.before(startDate)) {
            System.out.println(startDate);
        }

        if (date.equals(s) || s.contains(date.toString())) {
            if (date.compareTo(yesterday) > 1) {
                return;
            }
            System.out.println(lastYear.getTime());
        }
    } else {
        System.out.println(date.getTime());
    }
}
```

Figure 5.3: Most complex children element identified

Extract method

This is the step that an actual refactoring action is performed. The most complex children element identified from the previous step is extracted into a separate method as shown below in Listing 5.2. The extraction is performed using the IntelliJ built-in extract method action. After the extraction, it is obvious to see that the original method is now a lot easier to understand, all the complex low level details are moved into the **printCorrectDate()** method.

Compare cyclomatic complexity after refactoring

This is the last step in the refactoring process. The cyclomatic complexity of the method after refactoring is compared against its initial cyclomatic complexity. Both complexities are shown to the developer so that the impact on the method from the refactoring is revealed. This step is discussed in more details in Section 5.2.5.

```
public void complexMethod() {
    if (date.after(new Date()) || date.before(lastYear)) {
        printCorrectDate();
    } else {
        System.out.println(date.getTime());
    }
}

private void printCorrectDate() {
    if (date.before(tomorrow)) {
        System.out.println(date.getTime() - tomorrow.getTime());
    } else if (date.before(startDate)) {
        System.out.println(startDate);
    }

    if (date.equals(s) || s.contains(date.toString())) {
        if (date.compareTo(yesterday) > 1) {
            return;
        }
        System.out.println(lastYear.getTime());
    }
}
```

Listing 5.2: Most complex code block extracted into new method

5.2.2 Explanation of excessive cyclomatic complexity

As discussed before, it is important to provide the explanation of the code smell to the developers and help them to understand why the refactoring is needed and what are the benefits gained after the refactoring.

The explanation of the excessive cyclomatic complexity code smell is provided through the use of a dialog as shown in Figure 5.4

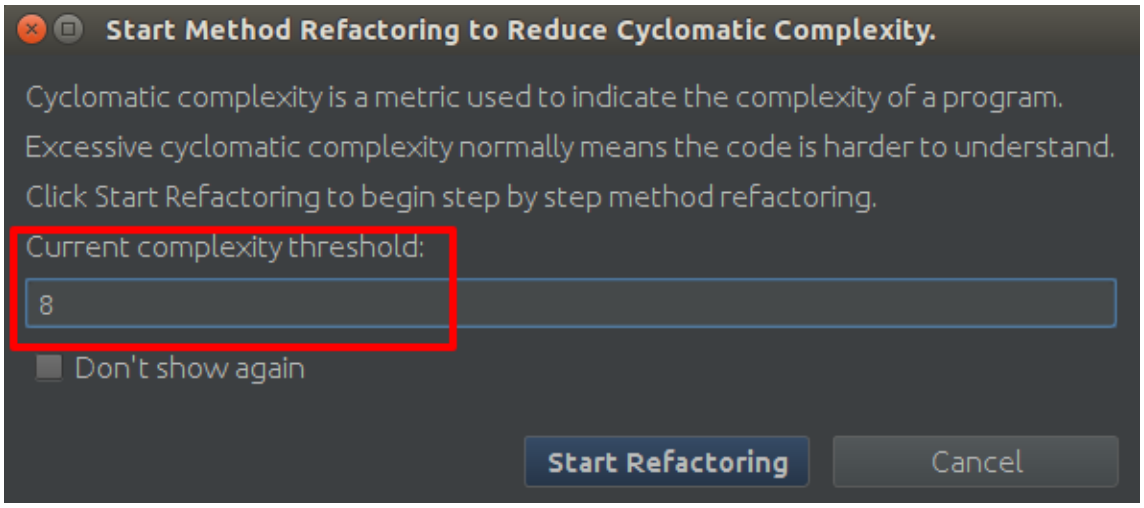

Figure 5.4: The first dialog displayed in the refactoring process

As this is the first step in the refactoring process, the dialog provides an option for the developers to cancel the refactoring process if they believe the code smell is not relevant after reading the explanations.

In addition, there are two other features included in this step. As shown in the dialog above, the current complexity threshold is displayed in a text field. Users are able to change the complexity threshold by editing the text field. Once the threshold is modified, the change to the threshold is reflected after the current refactoring process.

The second feature is the "Don't show again" check-box in the bottom left corner. By ticking the check-box, the explanation of the code smell is no longer displayed to the users. Although this feature might seem trivial, it greatly improves the user experience for developers who are already familiar with the code smell or might already go through the refactoring process several times. With the option of disabling the explanation dialog, they can skip this step and dive directly into the refactoring process.

Both of the two features make use of the IntelliJ built-in **PropertiesComponent**. The **PropertiesComponent** allows the plugin to store different key value mappings locally together with other local storage the IntelliJ IDEA uses. This utility is mostly used to store user preferences and customized configurations. For example, we make use of this utility to store the complexity threshold, the Listing 5.3 below shows how the threshold is updated when user changes the value in the text field.

```
textField.getDocument().addDocumentListener(new DocumentListener() {
    @Override
    public void insertUpdate(DocumentEvent e) {
        onChange();
    }

    @Override
    public void removeUpdate(DocumentEvent e) {
        onChange();
    }

    public void changedUpdate(DocumentEvent e) {
        onChange();
    }

    void onChange() {
        if (StringUtils.isNumeric(textField.getText())) {
            int newThreshold = Integer.valueOf(textField.getText());
                propertiesComponent.setValue(COMPLEXITY_THRESHOLD,
            newThreshold, DEFAULT_THRESHOLD);
        }
    }
});
```

Listing 5.3: Update PropertiesComponent when value of text field changes

5.2.3 Identify most complex element

The next step in the refactoring process is to observe the structure of the method and identify the most complex children elements inside the method. This is done by recursively calculating the cyclomatic complexity of the children elements and the element with the highest cyclomatic complexity is defined to be the most complex one.

Although the above strategy normally identifies the child element with highest cyclomatic complexity reliably, there are two special cases need to be handled. The first case is that more than one children elements might have the same cyclomatic complexity. In this case, the first

element among these elements is extracted.

The second edge case is that the method contains a child element that has the same cyclomatic complexity as the parent method. This could happen when:

1. The method has only a single direct child element. As shown below in Listing 5.4, the method has only one child element which is the if statement.
2. The method contains more than one direct children elements but one of them dominates over other elements and contains all the complexity as shown in Listing 5.5. Although there are three children elements in the method, the while statement contains all the complexity and it has the same complexity as the method.

```
public void complexMethod() {
    if (...) {
        ...
    } else {
        ...
    }
}
```

Listing 5.4: Method contains a single direct child element

```
public int complextMethod() {
    int i = 0;
    while (i < 10) {
        ...
    }
    return i;
}
```

Listing 5.5: Method contains a element that dominates all complexity

For both scenarios above, the approach we use is that we first identify the element which has the highest cyclomatic complexity (the same as the method complexity in this case), as we can not extract this element since it does not remove the code smell. Then we go inside this element and apply the same strategy to find the most complex element nested inside it. Finally, we extract the nested child element into a separate method. The flowchart 5.5 below illustrates the approach described.

After the element to be extracted is identified, a dialog is displayed to the user containing the code block to be extracted as shown in Figure 5.6 below. As the code block might be very long, we only show the start and end of the code block and omit the rest. The user should be able to locate the code block in the source code using the information provided.

5.2.4 Extract method

Following the code block is identified in the previous step, this step performs the extract method action. We make use of the IntelliJ built-in extract method action to extract the code block into a new method. The Figure 5.7 below shows the dialog displayed during the extract method action.

It is worth mentioning that, the method name *printCorrectDate* is entered manually instead of generating by the plugin. This is the one of the limitations when using the IntelliJ built-in

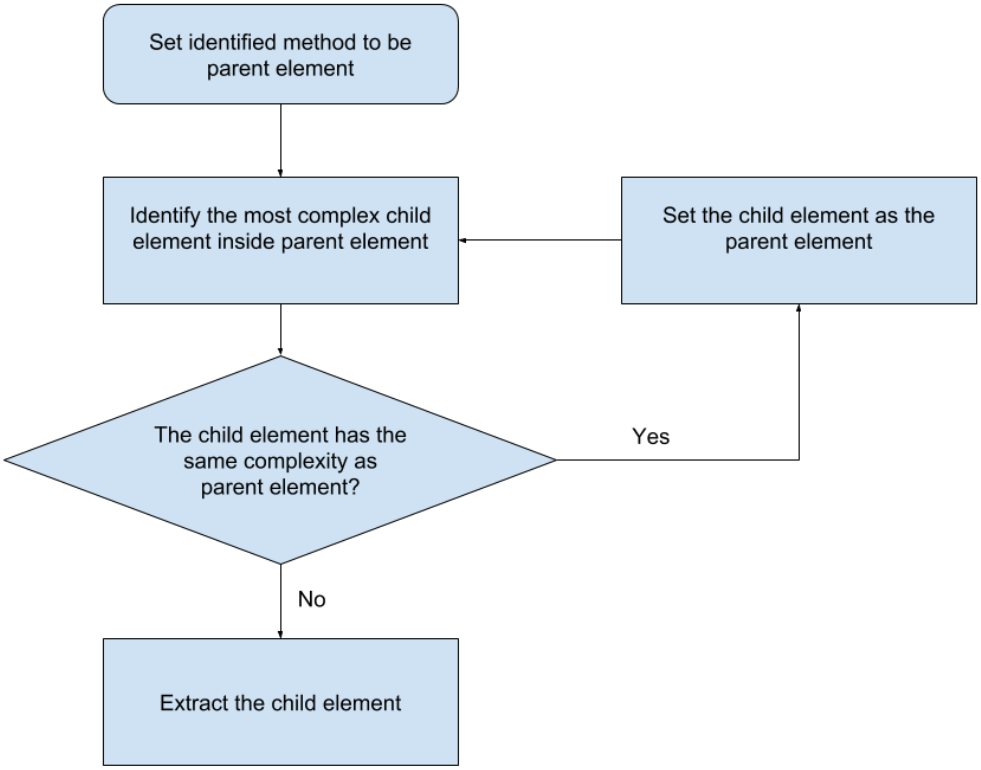

Figure 5.5: Approach to handle dominating child element

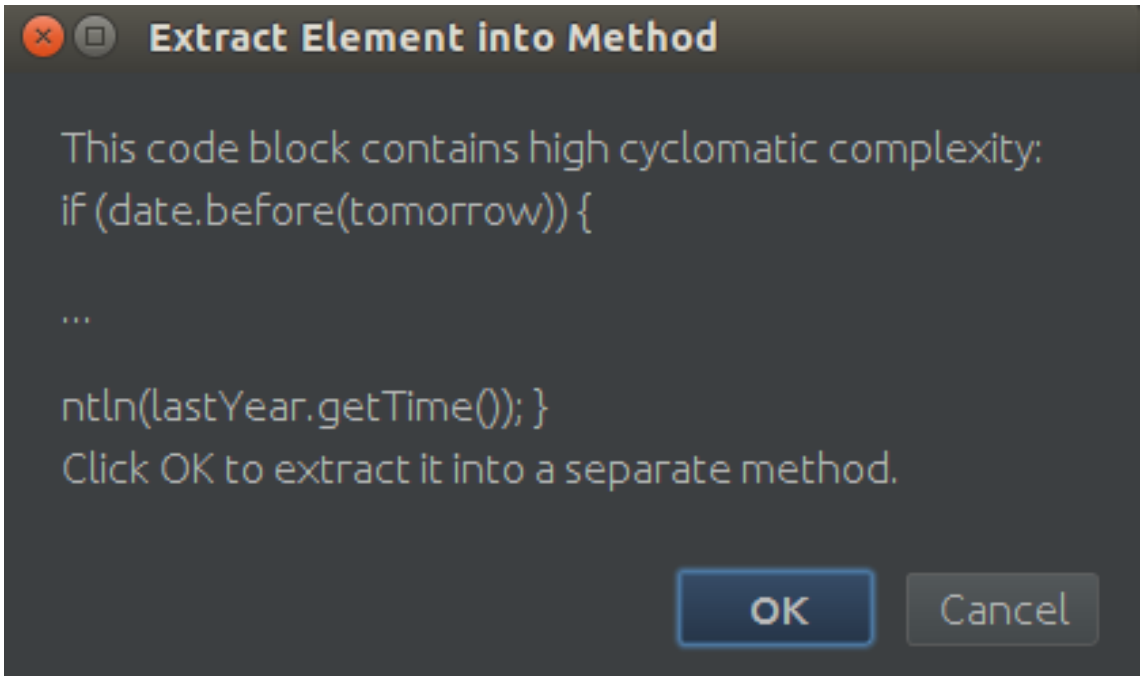

Figure 5.6: Dialog contains code block to be extracted

extract method action, as it restricts the ability to preset method name for the extract method. In addition, it is difficult to generate a meaningful name automatically as the name largely depends on the purpose of the extracted method which is not in the scope of this project.

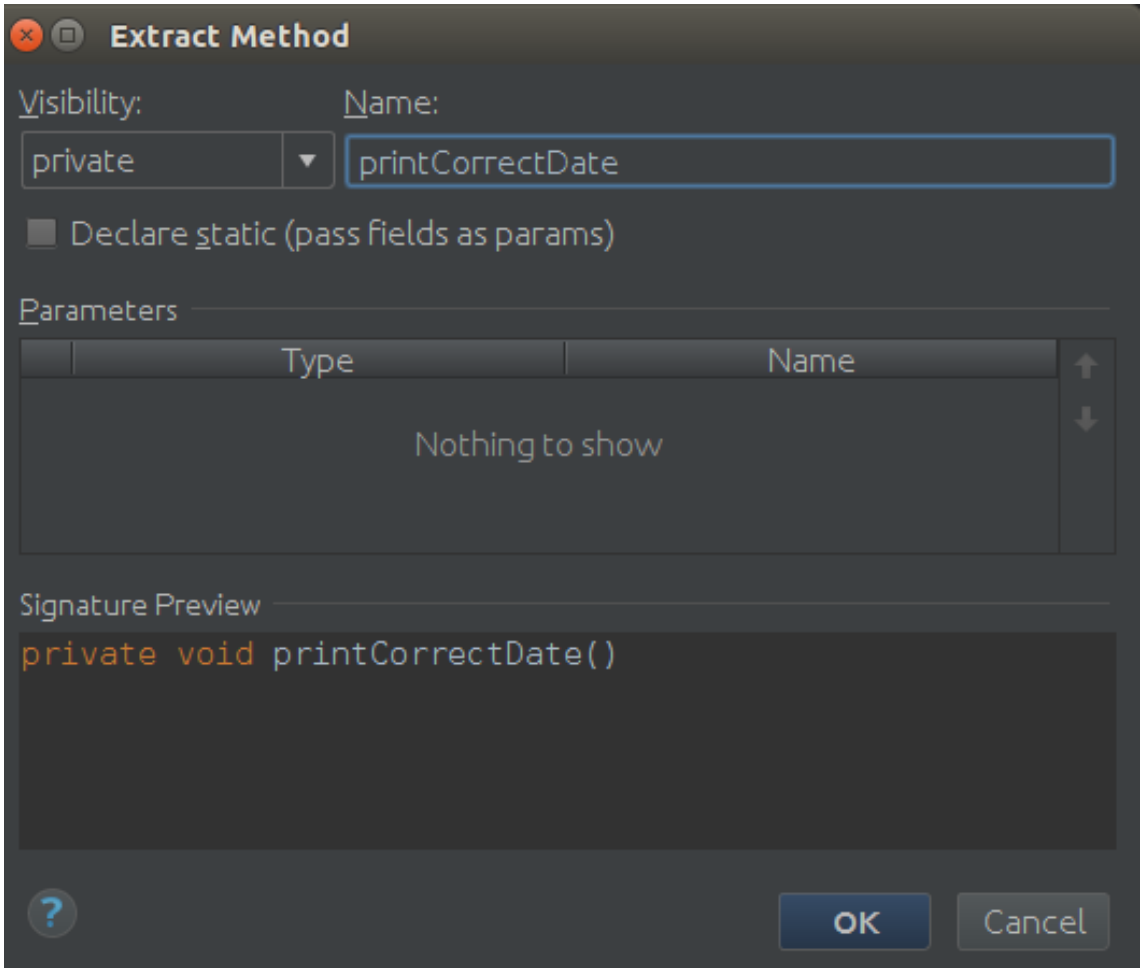

Figure 5.7: Extract method dialog

5.2.5 Compare cyclomatic complexity after refactoring

The last step in the refactoring process is to compare the cyclomatic complexity of the original method and the method after refactoring as shown in the dialog in Figure 5.8. The cyclomatic complexity of the original method is 8 and after extracting a code block that contains a lot of complexity into a new method, its new cyclomatic complexity is reduced to 3. The final outcome demonstrates that the refactoring process simplifies a method that suffers from excessive cyclomatic complexity code smell significantly. The example shown in the step by step walk-through in Listing 5.2 also shows that the method is now easy to read and understand.

5.3 Limitation

Although the excessive cyclomatic complexity inspection is able to achieve a high degree of accuracy in popular open source project for both detection and correction (see Chapter 7), there are edge cases not being covered and limitations associated with the existing techniques and methods. In the sections below, we will realize any limitations in detection and correction and discuss potential improvements to the current approach.

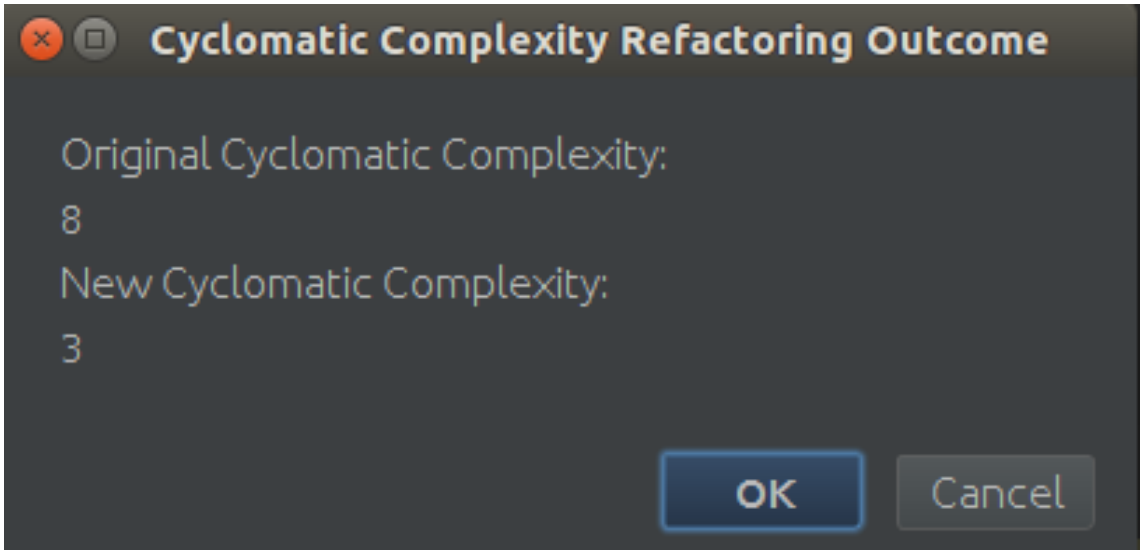

Figure 5.8: The outcome of excessive cyclomatic complexity refactoring

5.3.1 Detection Limitation

In the identification of the excessive cyclomatic complexity code smells, we make use of the metric based analysis. Although we are able to reliably detect any methods that have a cyclomatic complexity exceed the predefined threshold, we realize two main weaknesses of our approach.

The first weakness is that the detection relies heavily on the complexity threshold. In order to identify methods with excessive cyclomatic complexity, a well studied threshold is required to define the concept of "excessive". For example, when the complexity threshold is set to a low value (i.e. 1), almost all methods would be classified as containing code smell which is not the desired behavior. This weakness also increases the learning curve required for the plugin users. The user requires a good understanding of the cyclomatic complexity in order to define an appropriate threshold that performs well on the project he works on. Furthermore, some trial and error might be needed to test the performance of the code inspection under different threshold values.

To address this weakness, one of the solutions might be calculating the threshold value dynamically. This could be done by scanning and analyzing the whole project a developer is working on beforehand, and use heuristics to decide what a good default threshold is. However, this is outside the scope of this project.

The second weakness of the detection is that the code inspection focuses purely on the cyclomatic complexity of a method while there are other different metrics exist for code complexity evaluation. For example, the Lines-of-Code metric is one of the most common metrics used to evaluate method complexity. Although the code smell detection based on other metrics might be added to the plugin in the future, it is outside of the current scope of the project.

5.3.2 Correction Limitation

Although the correction of the excessive cyclomatic complexity code smell has a accuracy of more than 80% in the trails we carried out (See Section 7.3.2), there are still limitations on the refactoring techniques we choose.

The current strategy aims to identify a single element in the method that contains the highest cyclomatic complexity. However, in the example shown in Listing 5.6 below, multiple if statements have the same complexity. Although the current approach of extracting one of the if statements into a separate method reduces the cyclomatic complexity of the method, it would certainly be

more natural to extract all of them together. One of the improvements could be made to the correction strategy is to group multiple sibling elements and then extract them into a new method together.

```
public void complexMethod() {

    if (...) { ... } //cyclomatic complexity 4

    if (...) { ... } //cyclomatic complexity 4

    if (...) { ... } //cyclomatic complexity 4

    ...
}
```

Listing 5.6: Method contains multiple if statements each with same cyclomatic complexity

Chapter 6

Return Private Mutable Field Inspection

Although the "return private mutable field" code smell is the last code smell to be discussed, it is actually the first code smell to be tackled in this project. The code smell is first looked at because both the detection and correction focus on the return statement in a method, the scope is well defined and the required techniques are less complex compared to the two code smell inspections we discussed earlier (See Chapter 4, 5).

The detection and correction of this code smell is codified into the plugin as the *Return Private Mutable Field Inspection*. This chapter discusses the approaches used to identify the code smell, refactoring methods applied to eliminate the code smell, as well as any limitations associated with the chosen techniques.

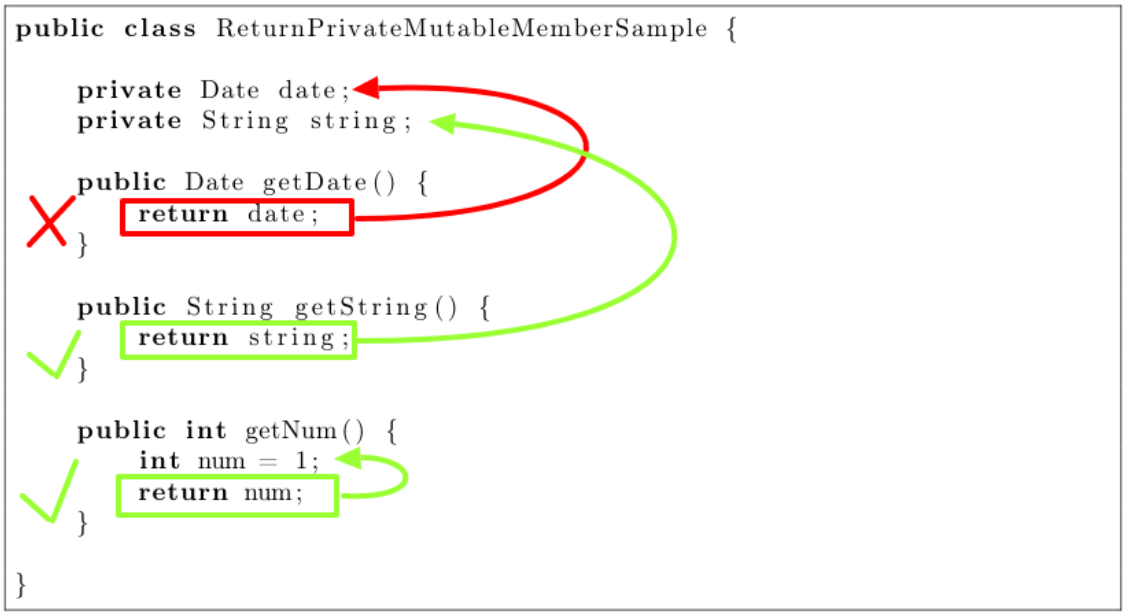

Figure 6.1: Procedure to examine switch statement

6.1 Identification

The most fundamental condition for the return private mutable field code smell is that it exists only in the return statement of a method, therefore we propose the following detection strategy:

1. Find all the return statements in the file currently edited by the developer.
2. Examine each of the return statements found, and determine whether it suffers the return private mutable field code smell

Figure 6.1 illustrates the examination of each of the return statements and determine if the expression being returned is a reference to a private mutable field. The first return statement returns "date" which is a reference to a private mutable object Date, so this return statement is classified as containing code smell. On the other hand, the two return statements below return a private immutable field (String is immutable) and a local variable respectively. Hence they are classified as not containing code smell.

Although we can determine quickly if the return expression is a private mutable field by just observing the code manually, the code inspection follows a number of rigorous steps to come up with the conclusion. The Figure 6.2 below illustrates the procedures followed to determine the code smell.

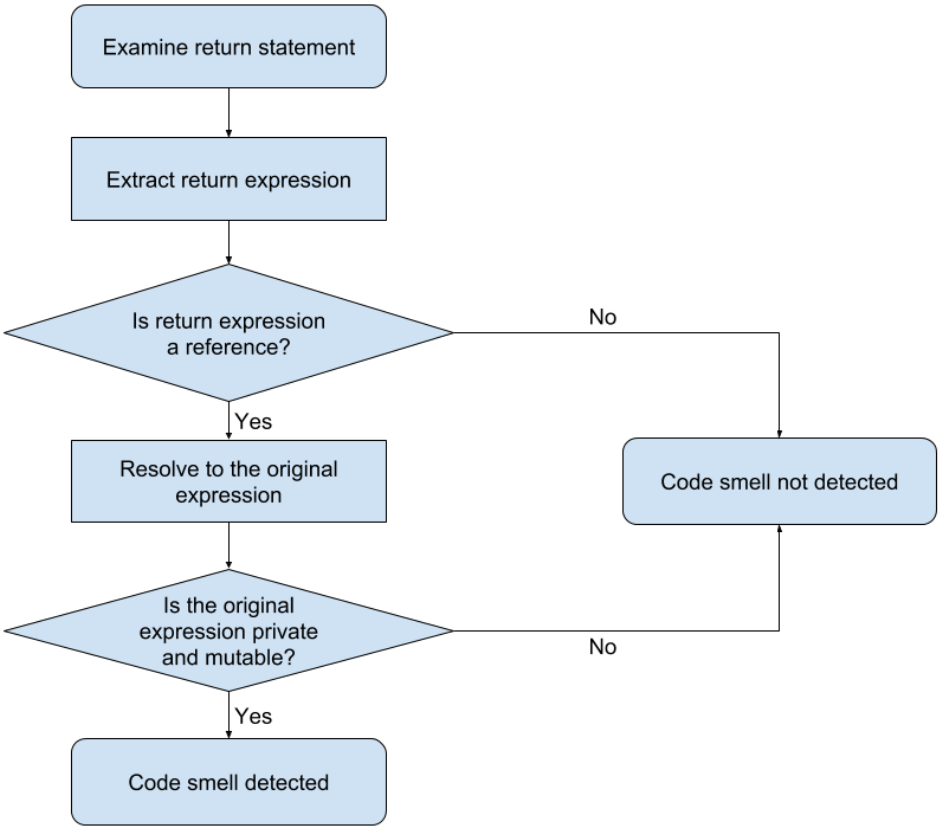

Figure 6.2: Identification procedure overview

During the detection procedures above, we make use of the APIs provided by the PSI elements extensively. For example, in order to resolve a reference to the original expression being referenced to, we make use of the **resolve()** method on the reference and it returns the original expression. To determine if the original expression is a class field is simple, we make use of the Java **instanceof** operator as below:

```
if (element instanceof PsiField) { ... }
```

Listing 6.1: Determine if an expression is a class field

Similar to the approach used to resolve a reference, for the validation on private field, the modifiers of the field is checked and the field is private if the list of modifiers contains the "private" keyword. The last validation step is to determine the mutability of the field. The mutability of the field can be determined using the PSI type associated with the object as shown in Listing 6.2.

```
boolean isImmutable = ClassUtils.isImmutable(field.getType());
```

Listing 6.2: Determine the mutability of a field

6.2 Refactoring

Once the code smell is detected in a return statement, developers are provided with the "Quick Fix" option to start the refactoring process. Unlike the "replace conditional with polymorphism" and "excessive cyclomatic complexity" code inspections, the refactoring for this code smell consists of only a single step.

We will first present a concrete example where a code smell in the return statement is detected and the refactoring outcome after applying the necessary transformations, follow by a detailed explanation of how the code is transformed step by step. Figure 6.3 shows the the highlighted return statement which suffers the return private mutable field code smell. A short description is displayed to explain the code smell to the developer.

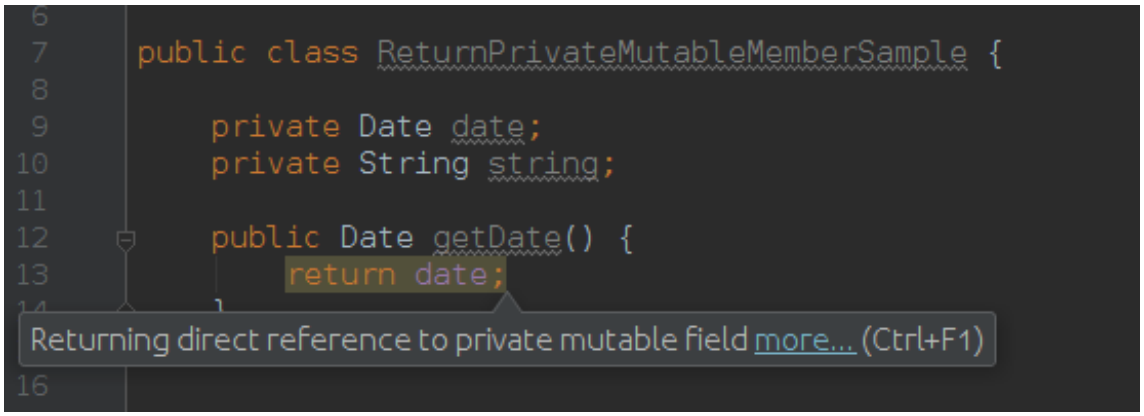

Figure 6.3: Code smell in return statement detected

After apply the refactoring to the code smell shown above, the return statement is modified into the version in Figure 6.4. The original return statement returns the private mutable field **date** from the method and allows the caller of the method to get hold of the reference to a private field. In order to mitigate this code smell, a defensive copy[27] is made to the **date** object and only the copy is returned. This prevents the caller of the method from holding the reference to a private mutable field and modify its state freely.

Although this seems to be a simple modification of the return statement, we will explain each of the underlying transformations conducted, and how each of the steps combined together to perform a robust and rigorous defensive copy.

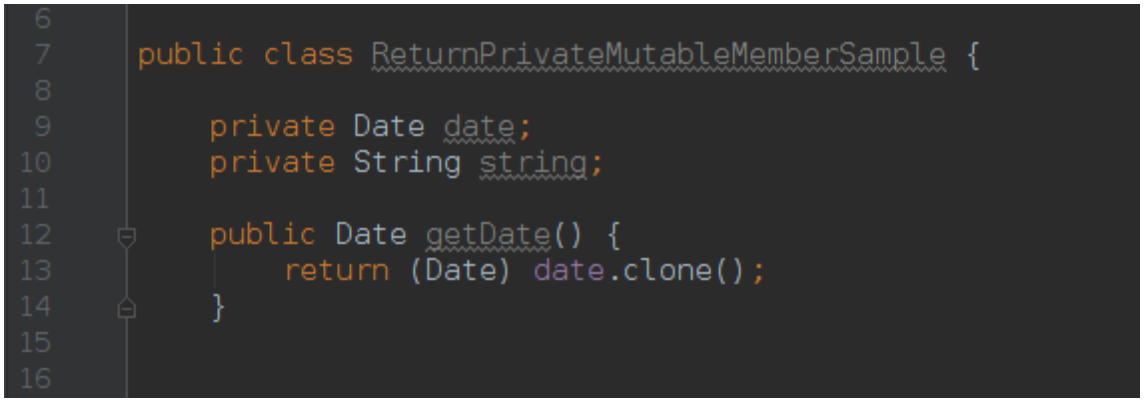

Figure 6.4: Code smell removed by making defensive copy

6.2.1 Clone object

The IntelliJ Platform provides an API called the **PsiElementFactory** to create PSI element from scratch. We make use of this mechanism to create a PSI element template and subsequently replace the template with actual objects. Listing 6.5 below illustrates how the an object clone template is created.

```
PsiElementFactory factory =
    JavaPsiFacade.getInstance(project).getElementFactory();

PsiMethodCallExpression cloneExpression = (PsiMethodCallExpression)
    factory.createExpressionFromText("a.clone()", null);
```

Listing 6.3: Create clone object template

6.2.2 Cast object clone

When an object is cloned, the cloned object is of the type **object** instead of the original class type. We need to cast the cloned object to the original type. This is also achieved using the same approach by creating a template of type casting and then replace the the **value** with the cloned object created from the previous step. To note that, not only the value is replaced, the casting type should also be replaced with the original class type as shown below.

```
PsiTypeCastExpression typeCast = (PsiTypeCastExpression)
    factory.createExpressionFromText("(Type) value", null);

typeCast.getCastType().replace(
    factory.createTypeElement(returnExpression.getType()));

typeCast.getOperand().replace(cloneExpression);
```

Listing 6.4: Cast object clone

6.2.3 Substitute return expression into template

The last step in the refactoring process is to substitute the original return expression with the template created from the previous steps as shown below. After this step, the original return statement is successfully transformed into a new return statement that makes a defensive copy of the return expression before returning it.

```
returnExpression.replace(typeCast);
```

Listing 6.5: Create clone object template

6.3 Limitation

The approach of creating templates and replacing each part of the template with real values helps to deal with different kinds of return expressions regardless whether the return expression is a literal, an object or a collection type. However, there exists a special case which requires some attention.

When making defensive copy to a collection such as a List or an object containing reference to another object, there are two common approaches. The first is to not only make a defensive copy of the collection, but also make a copy for each of the elements in the collection. This approach is also referred to as "deep copy"[28]. The second approach is to make copy of the collection without copy all the elements it contains, only the reference to the elements are duplicated. The second approach is referred to as "shallow copy".

In this code inspection, we make a design choice that shallow copy is used when making defensive copies. After careful considerations about the trade offs of the two approaches, we believe that making only shallow copy provides more flexibility to the developers and they are free to adopt the shallow copy with a proper deep copy if necessary.

Chapter 7

Evaluation

The evaluation of this project focuses on the assessments of the IntelliJ plugin being developed. Based on the objectives we outlined initially (see Section 1.2), we evaluate the plugin on the following aspects:

- Performance of code smell detections in terms of speed
- Performance of code smell detections in terms of accuracy
- Performance of code smell corrections in terms of accuracy and stability
- Usability of the plugin in practice

For each of the evaluation, we first illustrate the evaluation approaches being used, then we present the evaluation results.

7.1 Speed of Code Smell Detection

One of the objectives of this project is to provide a fast feedback loop to the developers while they work on the development task. We believe that the speed of code smell detections are crucial to achieve this object.

7.1.1 Evaluation Approach

To evaluate how long it takes to detect code smells in real world application source code, we run the code inspections on the open-source project *teammates* [29]. Teammates is a tool for managing peer evaluations and university students feedback paths currently used by hundreds of universities across the world. As this is a project with a large code base and being contributed to by a wide range of developers actively, we believe that this project represents a general real world project and would be a good candidate to conduct the speed evaluation on.

There are many different kinds of files included in the project, since the code inspections are only triggered on the file currently being edited by developers, we shortlist a subset of files that contain the most complex logic in the project and probably feature potential code smells. There are in total of 27 files under the shortlisted directory *"teammates/src/main/java/teammates/logic/"* and most of the files have more than 500 lines of code each. The code inspections are run on each of these 27 files, and the time taken for the code inspections to finish would be recorded.

```
public class InspectionTimeEvaluator {

    public static void start() { ... }

    public static void end() { ... }
}
```

Listing 7.1: InspectionTimeEvaluator developed for speed evaluation

Since the code smell inspections are run automatically, in order to produce a set of accurate and precise result, we implement an *InspectionTimeEvaluator* to record the time taken for code inspections to run on each file. The *InspectionTImeEvaluator* features two methods *start()* and *end()* as shown in Listing 7.1 below. An instance of the evaluator is injected into each of the code small inspections as shown in Listing 7.2. On start of the code inspection, a call is made to the *start()* method, a call is made to the *end()* method once it reaches the completion of the code inspection, and the total time taken would be recorded.

```
@Override
public void inspectionStarted(
    @NotNull LocalInspectionToolSession session,
    boolean isOnTheFly) {
    inspectionTimeEvaluator.start();
}

@Override
public void inspectionFinished(
    @NotNull LocalInspectionToolSession session,
    @NotNull ProblemsHolder problemsHolder) {
    inspectionTimeEvaluator.end();
}
```

Listing 7.2: InspectionTimeEvaluator is injected into code inspections

7.1.2 Evaluation Result

Table 7.1 below shows the time taken for inspections to run on each of the 27 files segmented by the three code smells inspections.

From the table we can see that the time taken for inspecting different files ranges from 10 milliseconds up to nearly 5000 milliseconds. In order to find out the why there is such a big gap when running inspections on different files, we look into the 10th file (took 10 milliseconds to run inspections) and the 4th file (took more than 1000 milliseconds for all three code inspections). We immediately find out the reason, the 10th file has only 5 lines code whereas the 4th file contains more than 2000 lines of code. The difference in the lines of code caused the significant difference in the inspection time.

Another interesting behavior we notice is that the three code smell inspections often took a very similar amount of time to run and even the same time sometimes. We believe that this behavior was caused by the underlying optimization IntelliJ Inspection System provides. Instead of running the code inspections one after another, IntelliJ Inspections System runs multiple code inspections all together in parallel in the background thread.

In order to gather a more constructive summary from the raw data, the number of files being

File	Replace conditional with polymorphism (millis)	Excessive cyclomatic complexity (millis)	Return private mutable field (millis)
1	2762	4769	2785
2	82	95	86
3	234	289	239
4	4690	1326	4708
5	44	45	45
6	298	894	300
7	378	946	391
8	87	87	92
9	211	293	295
10	10	10	11
11	530	562	563
12	67	74	75
13	707	718	734
14	50	51	51
15	542	848	852
16	350	763	853
17	644	999	1023
18	1023	334	1316
19	223	330	355
20	550	550	551
21	54	231	231
22	219	219	225
23	10	10	10
24	123	123	123
25	83	108	108
26	731	599	734
27	50	50	50

Figure 7.1: Code smell inspection speed evaluation results

inspected are aggregated into four time intervals based on the time taken for inspections to run. And the summary result is shown in table 7.2.

	1-100 milli	101-500 milli	500-1000 milli	1000-5000 milli
Replace conditional with polymorphism	10	8	6	3
Excessive cyclomatic complexity	8	8	9	2
Return private mutable field	8	9	6	4

Figure 7.2: Speed evaluation aggregated results

From the aggregated results, we can see that nearly 90% of the code inspections run under 1000 milliseconds, only 9 out of 81 code inspections run more than 1 second. This precisely matches the one of the objectives outlined for this project which is to provide a quick feedback loop to developers. With the majority of code inspections finish under 1 second, developers are warned about any code smells almost immediately.

7.2 Accuracy of Code Smell Detection

7.2.1 Replace Conditional With Polymorphism

To evaluate the detection accuracy of the code inspection and achieve a set of evaluation result that can represent a general case, we want to follow a similar approach as the evaluation for speed by running the code inspections on real world open source projects. However, due to the nature of this code smell, it is difficult to find a single project that contains a large amount of occurrences within a single project. We searched through the list of trending Java projects on GitHub [30], and any of these projects only contains a limited number of potential replace conditional with polymorphism code smells.

In order to conduct a good evaluation that covers many different edge cases possible, we short-listed a list of 30 switch statements from three different open source projects (Teammates, Elastic-

search [31], Apache Dubbo [32]). This is done by first searching through the code base and locate the switch statements. As the use of switch statement is the most fundamental condition that a using conditional instead of polymorphism code smell needs to satisfy. After that we manually review the switch statements and shortlist a subset of them as the dataset for the accuracy evaluation. The shortlisting process is to ensure that the set of switch statements cover a variety of edge cases in order to get a generalized evaluation. We selected 15 genuine switch statements that do not contain the code smell, combined with other 15 switch statements that feature the code smell together as the evaluation data.

The replace conditional with polymorphism code inspection is then run on each file that contains the shortlisted switch statement. We classify that a code smell is detected if a piece of code is highlighted in the file after the code inspection is run. Any other code inspections are disabled during the evaluation to ensure the code is only highlighted because it contains the using conditional instead of polymorphism code smell.

The table 7.3 below shows the results of the replace conditional with polymorphism detection accuracy evaluation.

	Code Inspection Positive	Code Inspection Negative
Manual Positive	15	0
Manual Negative	4	11

Figure 7.3: Replace conditional with polymorphism detection accuracy evaluation results

As we can see from the result, the plugin is able to detect all 15 code smell cases in the test sample. This shows that the code inspection covers the edge cases included in the test sample and able to detect them reliably. However, this might also suggest that with only 15 code smell examples, the test case is too small so that it is not possible to cover edge cases that the plugin might miss.

```
public static DayOfWeek resolve(String day) {
    switch (day.toLowerCase(Locale.ROOT)) {
        case "1":
        case "sun":
        case "sunday": return SUNDAY;
        case "2":
        case "mon":
        case "monday": return MONDAY;
        case "3":
        case "tue":
        case "tuesday": return TUESDAY;
        case "4":
        case "wed":

        ......
    }
}
```

Listing 7.3: Switch statement extracted from Elasticsearch project

On the other hand, there are 4 test cases where we do not think there is a code smell but wrongly detected by the plugin. One of the examples is shown in Listing 7.3. This piece of code is extracted from the Elasticsearch project, what it does is to convert a string into an **enum** that represents a day in a week. It is easy to see that this switch statement satisfies all the requirements we defined in the code inspection. For example, the control expression of the switch statement is a method call and the number of branches in the switch statement exceeds the default threshold

2. Hence the inspection classifies this switch statement as a code smell.

However, after reading and understanding the semantic of the code, it is obvious that the switch statement is used for converting one representation of a day in a week to a different representation. It is therefore not count as a code smell when we review it manually. This shows that the code inspection is prone to false positive. This weakness exists because the detection of the code smell is mainly based on the structure of the switch statement instead of a combination of both semantic meaning and static structure.

Based on these results, the replace conditional with polymorphism code inspection can detect 86% of the code smells correctly in the trials we carried out when classify switch statements as code smell based on their structure. Although it is highly possible that there are uncovered edge cases in a broader range of real world projects that would decrease the accuracy. We are also certain that some of the code smells detected by the code inspection might not be true as illustrated using the example above, developers might need to use their own judgment to decide in these situations.

7.2.2 Excessive Cyclomatic Complexity

The excessive cylomatic complexity code smell is a lot more common in real world projects, we therefore shortlist 50 methods from the Teammates project as the evaluation data. The set of methods are mixed with methods that have excessive cyclomatic complexity and methods that don't, there are 25 methods for each of these two kinds respectively. For methods that do not have excessive cyclomatic complexity, they are selected to meet at least one of the following properties:

- The method has a cyclomatic complexity just below the default threshold 8
- The method contains more than 30 lines of code
- The method has more than three nested levels

The above criteria are followed to ensure the evaluation data set contains a mixture of different methods and each of them is "complex" in some way. The evaluation would then be able to test the excessive cyclomatic complexity code inspection with these edge cases and reveal its weakness.

The table 7.4 below shows the result of the excessive cyclomatic code inspection detection accuracy evaluation.

	Code Inspection Positive	Code Inspection Negative
Manual Positive	25	0
Manual Negative	0	25

Figure 7.4: Excessive cyclomatic complexity detection accuracy evaluation results

The result above indicates a perfect detection of excessive cyclomatic code smell in the data set we used. This result is expected because the cyclomatic complexity has been well studied and can be calculated accurately as discussed in Chapter 5. This means that developers can have a high confidence about the code smells identified by the excessive cyclomatic complexity code inspection. However, we will show in later sections, the correction of the code smell has weaknesses and can be error prone under certain conditions.

7.3 Effectiveness of Code Smell Correction

In the evaluation of the code smell correction performance, we make use of the evaluation data collected from the detection evaluation. This has two main advantages: 1) It ensures that both

the detection and correction performance are evaluated on the same data set, which makes the evaluation result consistent; 2) The natural order of detecting a code smell and following by the correction of the code smell is also represented here. This simulates the order of actions a real user would perform and the evaluation result could demonstrate the real performance of the plugin when used by developers on real world projects.

In order to evaluate the performance of the correction, we define the following two criteria used for the evaluation:

- Whether the code still compiles after the refactoring
- Whether manual steps are needed to adjust the code
- Whether the code smells are still present

7.3.1 Replace Conditional With Polymorphism

When evaluate how well the replace conditional with polymorphism code inspection could fix the code smell through the step by step refactoring process, we use the 19 code examples being detected as code smells from the previous evaluation of the detection performance.

It is worth mentioning that, the 19 code examples not only contain 15 switch statements that contain code smells, but also 4 other switch statements which were not code smells but wrongly classified as code smells. We would like to observe and evaluate the behavior of the plugin when it applies the step by step refactoring to a piece of code that contains no code smell.

For each of the files contains the 19 switch statements, we open each of them and go through the step by step refactoring process. As we step through the process, if any one of the steps breaks the code, for example, the code no longer compiles, we stop and classify the refactoring process as a failure. If the refactoring process can be stepped through successfully, we then review the code to confirm if the code smell has been corrected. If the code smell still exists or further actions need to be made manually to improve the code, we do not classify the refactoring process as a failure, but instead we classify this as "manual step required".

The table 7.5 below shows the evaluation results.

	Failed to refactor	Manual steps required	Refactoring succeeded	Total
Correctly detected code smells	8	5	2	15
Wrongly detected code smells	4	0	0	4
Total	12	5	2	19

Figure 7.5: Detection of replace conditional with polymorphism evaluation results

From the results, we can see that all of the corrections for the wrongly detected code smells failed. This is not a surprise because the refactoring steps are designed to target the smelly switch statements. At each refactoring step, changes are made to the code to ensure they are in an intended shape for next steps to proceed. However, by forcing the changes to a normal switch statement, the code is broken at a early stage of the refactoring process.

For example, when applying refactoring steps to the code shown previously in Listing 7.3, the refactoring failed at the "creating subclasses" step. The refactoring process does not consider the case where there is no code inside a case branch and thus not able to move any code into the method in the subclasses. This eventually results in the failure of the refactoring process. Although this seems to be a failure of the refactoring step, we believe that this edge case should be addressed during the detection phase.

On the other hand, when looking at the correction evaluation for the correctly detected code smells, although there are only 2 out of 15 code smells being corrected automatically, this is because generally improvements to class and method names are required after the developer steps through

the refactoring process. When the developer is asked to name a method half way through the refactoring, the context might not be clear enough to produce a good name. Once the refactoring process is finished, the developer might have a better idea about naming the classes and methods. We do not consider these cases to be a failure in the refactoring process, therefore the refactoring process can be went through without breaking existing code in 7 out of 15 cases. We achieve an accuracy of about 45% in the code smell correction phase.

This accuracy is not as high as expected. This is mainly due to the fact that there are many edge cases in a real world project, and along the seven refactoring steps, it was difficult to cover all the edge cases when developing the refactoring process against a simpler example. Out of the 8 cases where the refactoring failed at one of the steps, their failures can be categorized into three situations. One of the situation is that it does not handle the a empty case branch as illustrated above. The second situation is that the case value in the case branch is not expected as shown in the Listing 7.4 below. During the "creating subclasses" step, we improved the naming of subclasses a number of times to handle more and more edge cases. However, the refactoring step is still not able to name the subclasses correctly when the case value contains special characters such as "." in the *"JSONToken.COMMA"* example below.

```
switch (token.type) {
    case JSONToken.COMMA:
        ...
    case JSONToken.NULL:
        ...
    case JSONToken.BOOL:
        ...
}
```

Listing 7.4: Switch statement extracted from Apache Dubbo project

Another situation that causes the refactoring process to fail is not just within the switch statement itself, but related to the bigger structure it belongs to. When implementing the refactoring steps, an assumption was made that the switch statement is inside a class. However, the examples from real world projects show that this assumption is too strong. Listing 7.5 shows that when the switch statement is inside an enum, the "creating subclass" step failed straightaway due to the fact that it tries to create subclasses extending the enum.

From the evaluation results, we can see that although the refactoring works well on many cases, there are several edge cases that were not handled by the refactoring steps. Although it is showed that some future work is required to handle the edge cases discovered in order to improve the stability and accuracy of the code inspection, we believe that the step by step refactoring process provides a certain amount of assistance to the developers in the code smell correction. Even though it might fail at one of the steps, it acts as a good pointer to allow developers to discover the underlying problem in the code smell.

```
public enum ActionExecutionMode {

    ...

    public static ActionExecutionMode resolve(String key) {

        ...

        switch (key.toLowerCase(Locale.ROOT)) {
            case "simulate":        return SIMULATE;
            case "force_simulate":  return FORCE_SIMULATE;
            case "execute":         return EXECUTE;
            case "force_execute":   return FORCE_EXECUTE;
            case "skip":            return SKIP;
        }
    }

    ...
}
```

Listing 7.5: Switch statement inside enum

7.3.2 Excessive Cyclomatic Complexity

Similar to how the evaluation of replace conditional with polymorphism evaluation was carried out, we use the same data set from the detection evaluation and conduct the correction evaluation on the 25 methods that was identified to have excessive cyclomatic complexity.

The table 7.6 below shows the result of excessive cyclomatic complexity correction evaluation.

Failed to refactor	Manual steps required	Refactoring succeeded	Total
4	7	14	25

Figure 7.6: Correction of excessive cyclomatic complexity evaluation results

From the results, we can see that the excessive cyclomatic complexity code smell in 14 out of 25 methods can be fixed automatically. A further 7 methods can be improved using the refactoring guidance provided by the plugin. Out of the 7 cases, we observe two main reasons why manual steps are needed to improve the code:

- Manual steps conducted to rename the extracted method. As the develops go through the refactoring process provided by the plugin, some code from the original method is extracted into a new method and they are asked to name the new method. At that point, the developers do not have a good knowledge about the new method and it is likely that the name given does not represent what the method does. After the refactoring steps are stepped through, developers now have a better context and a different name for the method might be more appropriate and hence it requires them to manually modify the name of the method. To improve the user experience in these situations, an additional step might be added to the end of the refactoring process to allow developers to change the method name (again). This step could be easily added in the future.
- Another situation where manual steps are required is that the original method still contains the excessive cyclomatic complexity code smell. This happens because the original method might have a very high cyclomatic complexity, extracting part of it into a new method might result in three possibilities: 1) The original method is still complex; 2) The extracted new

method is complex; 3) Combination of the first two possibilities - both methods are complex. Although the excessive cyclomatic code smell still exists, the complexity of the methods are decreased. And the developers are encouraged to apply the refactoring steps again to further reduce the cyclomatic complexity. This behavior was also intended because it allows developers to realize the whole process of reducing the cyclomatic complexity of a method.

Both of the two situations above are expected, so we consider the refactoring to a total of 21 out of 25 methods to be successful. Therefore, we achieve an accuracy of 84% from the data set we use. Based on the results, we believe that the automatic correction of excessive cyclomatic complexity code smell is very effective in practice.

Although it is showed that the refactoring is effective, it is important to understand the cases where the refactoring failed. The Figure 7.6 below shows a code example where the automatic refactoring failed to remove the code smell.

```
public UserType getCurrentUser() {
    User user = getCurrentGoogleUser();

    if (user == null) {
        return null;
    }

    UserType userType = new UserType(user);

    if (isAdministrator()) {
        userType.isAdmin = true;
    }

    if (isInstructor()) {
        userType.isInstructor = true;
    }

    if (isStudent()) {
        userType.isStudent = true;
    }

    return userType;
}
```

Listing 7.6: Before refactoring

After stepping through the automatic refactoring process, a new method **validateUser()** is extracted from the **getCurrentUser()** method as shown in Figure 7.7 below. Although some code has been extracted out of the method, the cyclomatic complexity of the original method is not reduced. This is due to the fact that in order to maintain the semantic of the method, a new if statement is introduced to wrap the call to the extracted method, which results in no change in the cyclomatic complexity after the refactoring. In these situations, although the existing still compiles, the refactoring process does not have any impact on the method cyclomatic complexity. We therefore consider these cases to be failures in the code smell correction evaluation.

```
public UserType getCurrentUser() {
    User user = getCurrentGoogleUser();

    if (validateUser(user)) return null;

    ...
}

private boolean validateUser(User user) {
    if (user == null) {
        return true;
    }
    return false;
}
```

Listing 7.7: After refactoring

7.4 User Testing

One of the objectives in this project is to allow developers from a wide range of experiences to be able to use the tool developed and benefit from using it. We put a lot of effort to ensure users are deeply involved in the development process of the plugin and their valuable feedback are used to improved the plugin.

7.4.1 Usability Testing

In order to gather feedback from developers with a wide range of experiences, we participate in the "Project Fair" organized by the Computing Department to conduct usability testing. The goal of the usability testing is to find out if the plugin is easy to understand, steps can be followed intuitively and also discover any UI issues that are easily ignored while developing the plugin.

We interviewed 7 students in total ranging from Year 1 JMC student to Year 4 computing student. We intended to interview more students, however, the time taken for one student to go through the code smell detection and correction process is significantly longer than expected. Although it is believed that it would be better to receive one deep insight and constructive feedback than many less insightful suggestions.

During the interview process, we try not to ask direct feedback from the students, but instead we observe their behavior while they interact with the plugin and gather feedback from the observations. There are a number of interesting insights:

- Although the code inspections run automatically and code block that contains a code smell is highlighted, Year 1 students are struggled to start the refactoring process. In order to start the refactoring process, developers need to enter the "Quick Fix" mode using the shortcut "Alt + Enter" in IntelliJ IDEA. However, a number of students are not familiar with this shortcut. This finding is really insightful since we did not experience this difficulty before. However, due to the fact that the code inspections are built on top of the IntelliJ Inspections System, it is difficult to start the refactoring process using other methods.

- Another suggestion from a Year 4 student is that the naming of subclasses in the replace conditional with polymorphism code inspection should be improved. This issue is discussed in Section 4.2.5. This is the first time this issue being brought up and we hence reiterate and

improve the approach used to generate class and method names. The issue is fixed by our subsequent actions.

While the students are going through the refactoring process of the code inspections, the time taken for them to finish the refactoring process and remove code smell successfully is recorded. In addition to go through the refactoring process with the assist of the plugin, they are also asked to do the refactoring without the help of the plugin. The time taken for a student to go through the refactoring process manually ranges from two minutes up to about ten minutes. Some of them failed to finish the refactoring due to their unfamiliarity with the code smell.

When looking at the time taken to go through the refactoring process using the step by step guidance provided by the plugin, we notice that the refactoring generally takes less than one minute. This significant difference in the time taken shows that the plugin not only assists developers to fix code smells, but also saves developer's time and effort and increases developer's productivity.

7.4.2 Open Source Contribution

As the plugin is published to the JetBrains Plugins Repository, the download and use of the plugin is open to the public. There are 40 downloads of the plugin to date as shown in the screenshot below in Figure 7.7.

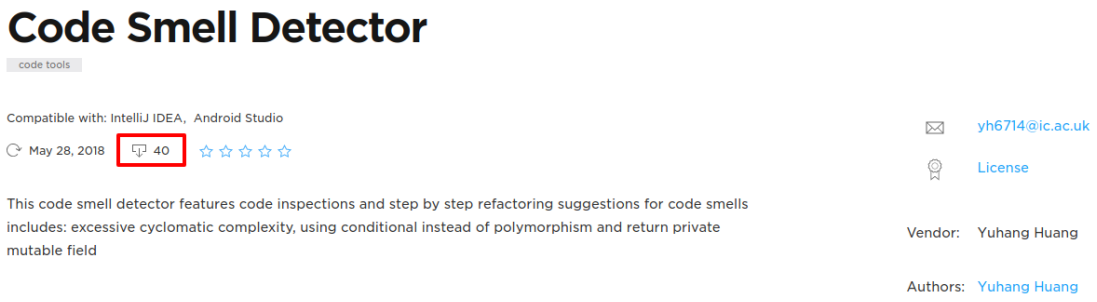

Figure 7.7: Plugin page on JetBrains Plugins Repository screenshot

We also open sourced the plugin so that developers from a wide range of background could give feedback and contribute to the plugin development. These kind of feedback would expect to be very different from the feedback gathered from students in the project fair.

The figure 7.8 below shows an example of the feedback submitted as a GitHub issue from a developer that is interested in this plugin.

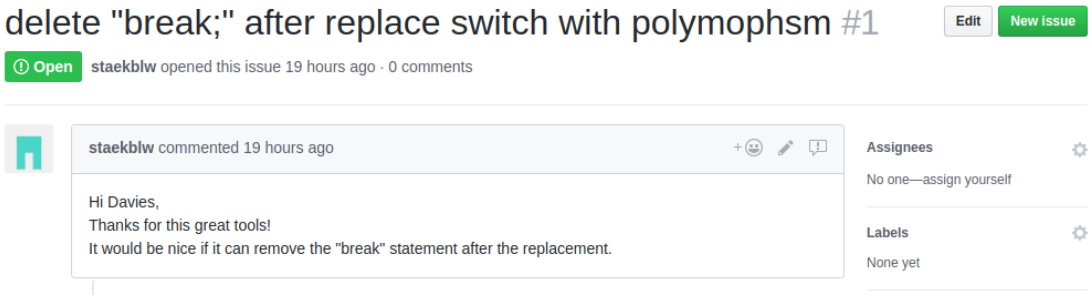

Figure 7.8: A suggestion submitted as a GitHub issue

Chapter 8

Conclusion

In this project we develop an IntelliJ plugin that assists developers in the detection and correction of three different code smells in Java code. "Replace conditional with polymorphism", "Excessive cyclomatic complexity" and "Return private mutable field" code smells are detected quickly and accurately. Automatic correction consists of step by step refactoring guidance is provided to help developers through the process of removing these code smells.

8.1 Achievements

Fast Detection of Code Smells

Code inspections are run continuously in the background while a developers is making changes to the code, the developer is warned about any code smells almost immediately. From the evaluation we carried out, out results indicate that 90% of the code inspections complete under 1 second. This is achieved by conducting local inspecting on the file that is currently editing by the developer. By providing fast feedback to a developer, quick adjustments and improvements could be made instantly and the amount of time and effort spent to remove the code smell is minimized.

Accurate Detection of Code Smells

The three code smells tackled in this project are not only detected quickly but also accurately. By analyzing switch statement structures, using conditional instead of polymorphism code smell can be detected in 86% of cases in the evaluation being carried out. Using metric based analysis, methods with excessive cyclomatic complexity code smells are also detected highly reliably in popular open-source projects.

Effective Correction of Code Smells

Once a code smell is detected, automatic correction consists of step by step refactoring guidance is provided to help developers through the process of removing these code smells. Small refactoring actions are performed at each step and code structures are validated before proceeding to the next step, which makes the code smell refactoring process stable and effective.

By performing automatic correction on code smells found on popular open-source projects, the evaluation being carried out shows the success rate of 45% and 84% on the correction of replace conditional with polymorphism code smell and excessive cyclomatic complexity code smell respectively. From the observations during user testing, the refactoring guidance provided reduces

the time taken to remove a code smell, from in average 5 minutes to under 1 minute. All of the evidence indicates that the automatic correction is sufficiently effective to be used in practice, it saves developer's time and effort, and significantly increases developer's productivity.

Other Achievements

In addition to the automatic detection and correction of code smells, the plugin also specifically provides in-depth explanation to code smells and detailed instructions during the refactoring process. This helps developers to have a better understanding of the code smells and improve their ability writing good quality code.

8.2 Future Work

Although the main objectives outlined initially have been successfully met and the performance of the plugin is shown to be satisfactory when used in practice, there are areas of improvements could be made to the plugin in the future. Some of the improvements are proposed in the Limitation Sections 4.3, 5.3, 6.3, other possible future work is summarized below:

Handle Refactoring Edge Cases

The evaluation of the automatic correction has shown that there are many edge cases are not covered currently. In order to improve the accuracy and stability of the code inspections, a wider range of edge cases should be handled correctly by the plugin.

Support More Types of Complexity Analysis

The currently supported cyclomatic complexity analysis for code smell detection has been shown to be effective and reliable, it would add more value to the plugin if there are more types of complexity analysis could be used for a more comprehensive code smell detection.

Support for Extension

The IntelliJ Platform provides a system called "Extension Points" to allow the plugin to be extended with more functionalities. The support for extension is outside the current scope of the project, but adding the support in the future would provide great flexibility to the users of the plugin to tackle other code smells using different techniques.

Bibliography

[1] Checkstyle. http://checkstyle.sourceforge.net/. Accessed: 1 June 2018.

[2] S. C. Johnson. Lint, a c program checker. *Unix Programmer's Manual*, 2(15):292–303, 1978.

[3] M. Fowler. Refactoring: Improving the design of existing code. *Addison-Wesley*, 1999.

[4] Software quality. https://en.wikipedia.org/wiki/Software_quality. Accessed: 1 June 2018.

[5] F.; Bavota G.; Oliveto R.; Di Penta M.; De Lucia A.; Poshyvanyk D. Tufano, M.; Palomba. When and why your code starts to smell bad. *IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, 2015.

[6] Tell-don't-ask principle. https://martinfowler.com/bliki/TellDontAsk.html. Accessed: 1 June 2018.

[7] Cyclomatic complexity. https://en.wikipedia.org/wiki/Cyclomatic_complexity. Accessed: 1 June 2018.

[8] H. Watson; Thomas J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST Special Publication 500-235*, 1996.

[9] Defects prediction. http://www.eecs.qmul.ac.uk/~norman/papers/defects_prediction_preprint105579.pdf. Accessed: 1 June 2018.

[10] Java string. https://docs.oracle.com/javase/7/docs/api/java/lang/String.html. Accessed: 1 June 2018.

[11] Checkstyle wikipedia. https://en.wikipedia.org/wiki/Checkstyle. Accessed: 1 June 2018.

[12] Findbugs. http://findbugs.sourceforge.net/. Accessed: 1 June 2018.

[13] Spotbugs. https://spotbugs.github.io/. Accessed: 1 June 2018.

[14] Jdeodorant. https://github.com/tsantalis/JDeodorant. Accessed: 1 June 2018.

[15] Vignelli. https://www.doc.ic.ac.uk/teaching/distinguished-projects/2015/s.stuckemann.pdf. Accessed: 1 June 2018.

[16] Quick fix. https://www.jetbrains.org/intellij/sdk/docs/tutorials/custom_language_support/quick_fix.html. Accessed: 1 June 2018.

[17] Developer survey results. https://insights.stackoverflow.com/survey/2017#most-popular-technologies. Accessed: 1 June 2018.

[18] Intellij idea. https://www.jetbrains.com/idea/. Accessed: 1 June 2018.

[19] Program structure interface. https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi.html. Accessed: 7 June 2018.

[20] Fagan inspection. https://en.wikipedia.org/wiki/Fagan_inspection. Accessed: 1 June 2018.

[21] Intellij general threading rules. https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/general_threading_rules.html. Accessed: 7 June 2018.

[22] Dialogwrapper. https://www.jetbrains.org/intellij/sdk/docs/user_interface_components/dialog_wrapper.html. Accessed: 7 June 2018.

[23] Incremental build model. https://en.wikipedia.org/wiki/Incremental_build_model. Accessed: 8 June 2018.

[24] Intellij platform sdk devguide. http://www.jetbrains.org/intellij/sdk/docs/basics/testing_plugins.html. Accessed: 8 June 2018.

[25] Sikuli. http://www.sikuli.org/. Accessed: 8 June 2018.

[26] R. C. Martin. Clean code: A handbook of agile software craftsmanship. *Upper Saddle River, NJ, USA: Prentice Hall PTR*, 2008.

[27] Defensive copying. http://www.javapractices.com/topic/TopicAction.do?Id=15. Accessed: 1 June 2018.

[28] Difference between shallow copy vs deep copy in java. http://javaconceptoftheday.com/difference-between-shallow-copy-vs-deep-copy-in-java/. Accessed: 1 June 2018.

[29] Teammates. https://github.com/TEAMMATES/teammates. Accessed: 8 June 2018.

[30] Trending java projects. https://github.com/trending/java. Accessed: 8 June 2018.

[31] Elasticsearch. https://github.com/elastic/elasticsearch. Accessed: 8 June 2018.

[32] Apache dubbo. https://github.com/apache/incubator-dubbo. Accessed: 8 June 2018.