

**Imperial College  
London**

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND  
MEDICINE

DEPARTMENT OF COMPUTING

MENG SOFTWARE ENGINEERING

---

# Visualising Connascence to Drive Refactoring

---

*Author:*  
Radu GHEORMAN

*Supervisor:*  
Dr. Robert CHATLEY

*Second Marker:*  
Dr. Mahdi CHERAGHCHI

SUBMITTED IN PART FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MENG  
SOFTWARE ENGINEERING

June 21, 2017



## Abstract

We present *Connascer*, an IDE plugin which helps programmers improve the quality of their code by providing real-time detection of code problems that may negatively impact the long-term velocity to change of software, as well as by offering solutions in the form of automatic code refactorings at the source of such problems to assist developers in the process of eradicating them.

Our plugin is relevant because it is the first such attempt to support detection of connascence, a software quality metric that is associated with those components that must change together in order to preserve overall system correctness. Unlike other code metrics, connascence considers each problem through more than one dimension, which is why we are able to quantify the potential effect within the application of each problem, and thus empower developers with knowledge about which issues need to be prioritised for refactoring.

Our results show that we are able to offer detection for supported types of connascence, namely connascence of meaning and connascence of position, with more than 90% accuracy, and successfully refactor detected problems in more than 80% of the cases. Additionally, to date more than 90 developers have reviewed and tested *Connascer*.



## Acknowledgements

I would like to thank my supervisor, Dr. Robert Chatley, for his invaluable advice and guidance throughout the entire project. His contributions have been decisive in crafting and bringing *Connascer* to life.

I would like to thank Kevin Rutherford for sharing his incredible knowledge about connascence and how to fix it.

I would like to thank my personal tutor, Dr. Maria Valera-Espina, for helping me throughout the past four years with useful academic advice, as well personal.

Finally, to my family and friends, for their unconditional support and encouragement that highly helped me throughout my whole degree and beyond that.

# Contents

<b>Contents</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Motivation . . . . .	3
1.2 Relevance . . . . .	4
1.3 Objectives . . . . .	6
1.4 Contributions . . . . .	6
<b>2 Background</b>	<b>7</b>
2.1 Static Checkers . . . . .	7
2.1.1 IntelliJ IDEA . . . . .	7
2.1.2 Checkstyle . . . . .	9
2.1.3 FindBugs . . . . .	10
2.1.4 PMD . . . . .	11
2.1.5 Others . . . . .	12
2.2 Code Metrics . . . . .	12
2.3 Unified Modelling Language (UML) . . . . .	13
2.4 Proposed Solution . . . . .	14
2.4.1 Connascence . . . . .	14
2.4.2 Connascence Strength . . . . .	15
2.4.3 Connascence Locality . . . . .	19
2.4.4 Connascence Visualisation . . . . .	20
2.4.5 Connascence Degree . . . . .	21
2.4.6 Real-time analysis and Refactorings . . . . .	22
2.4.7 Final remarks . . . . .	22
<b>3 Implementation Design</b>	<b>23</b>
3.1 Problem detection . . . . .	23
3.1.1 Source code parsing . . . . .	23
3.1.2 Detection algorithms . . . . .	25
3.2 Problem visualisation . . . . .	26
3.3 Problem refactoring . . . . .	28
<b>4 Connascence of Meaning</b>	<b>30</b>
4.1 Identification . . . . .	31
4.1.1 Binary Expressions . . . . .	31
4.1.2 Switch Statements . . . . .	35
4.1.3 Return Statements . . . . .	35

4.1.4	Methods Calls . . . . .	37
4.2	Refactoring . . . . .	38
4.2.1	Extract literal . . . . .	38
4.2.2	Wrap expression inside Optional . . . . .	39
4.2.3	Wrap method return type with Optional . . . . .	39
<b>5</b>	<b>Connascence of Position</b>	<b>43</b>
5.1	Identification . . . . .	44
5.1.1	Methods and Constructors . . . . .	44
5.1.2	Methods Calls . . . . .	45
5.1.3	“new” Expressions . . . . .	48
5.2	Refactoring . . . . .	48
5.2.1	Introduce Parameter Object . . . . .	49
5.2.2	Replace Constructor with Builder . . . . .	51
<b>6</b>	<b>Evaluation</b>	<b>54</b>
6.1	Accuracy . . . . .	54
6.1.1	Evaluation method . . . . .	54
6.1.2	Connascence of Meaning . . . . .	54
6.1.3	Connascence of Position . . . . .	57
6.2	Performance . . . . .	62
6.2.1	Evaluation method . . . . .	62
6.2.2	Evaluation results . . . . .	63
6.3	Implementation Design . . . . .	64
6.4	Usability . . . . .	65
6.4.1	User feedback . . . . .	65
6.4.2	Plugin monitoring . . . . .	67
6.5	Remarks . . . . .	69
<b>7</b>	<b>Conclusion</b>	<b>70</b>
7.1	Achievements . . . . .	70
7.2	What we would do differently next time . . . . .	71
7.3	What we have learnt . . . . .	72
7.4	Future work . . . . .	72
	<b>Bibliography</b>	<b>73</b>

# Chapter 1

## Introduction

Improving the quality of code has always been a topic of interest for programmers, and throughout the time there have been many proposals to help programmers achieve this. Early such attempts can be traced back to when the subroutine was invented, as it helped reduce code duplication and enhanced its reusability [1]. In more recent years, a focus in the programming industry has been on developing tools to automate the detection process of problems that may lead to a decrease in the quality of code.

A large part of the built tools are aimed at finding problems within source code without executing it, a technique known as *static program analysis* [2]. Depending on the complexity of the problems they try to find, some tools may require a broader knowledge of the application than some other tools that may limit their detection algorithms to narrower scopes. Advantages of static analysis tools include that they are capable of detecting very complex problems, that are otherwise hard to find by developers without additional aid. On the other hand, one of the common disadvantages is that depending on the complexity of detected problems, the feedback loop is often quite slow, as a considerable amount of time may be required for analysis.

Furthermore, code metrics are another technique used by programmers to quantify the quality of their code [3]. Ranging from relative simple concepts such as the number of lines of code to more complex measures like component cohesion, these are tools that often provide relatively quick feedback in terms of a number. For example, a component may have 100 lines of code, while on the scale of cohesion, a component may score in the upper range, meaning that it is highly cohesive. The benefit with these metrics is that they are normally relatively fast to compute, but the downside is that sometimes a single number does not tell the whole story behind code, thus such a mapping may be misleading.

Although they vary in technique applied, and potentially even the nature of the underlying analysis, all these different solutions are attempts to help programmers deliver better products by improving the quality of their code. Certainly no one tool alone is the definite answer in designing quality code. It is why we decided to provide users with yet another solution, aimed not only at improving on the downsides of existing solutions, but also at delivering information that is easier to interpret and simpler to act upon, while also providing real-time analysis.

### 1.1 Motivation

Derived from Latin, *connascence* means having been born together, which, in programming terms, can be interpreted as software components that have evolved to a state where a change in one would require the other to be modified in order to preserve the overall system correctness. This coupling



relationship is thought to be detrimental to software, as it often hinders the ability of software to evolve. It is for this reason why connascence is perceived as a software quality metric and taxonomy for different types of coupling [4].

“As the real world changes, so too must our code” [5]. In fact, change is thought to be a fundamental aspect of code, and Sandi Metz notes that “changing requirements are the programming equivalent of friction and gravity” [6]. Indeed, the question of change in software is a matter of when it will be required, rather than if it will be required. It is not unseen that software which undertakes several changes becomes what is known as “big ball of mud” [7] - the state where further change is so hard to incorporate, that developing a new project altogether would be more advantageous [8].

Connascence is referred to as software quality metric because it primarily identifies the flexibility of code. Keeping a low friction to change “is essential for maintaining long-term development velocity” [5]. The added benefit of connascence is that it need not be computed after the whole software has been written, but rather on the go, as every line of code is being crafted. It is therefore in the interest of every programmer to be aware of their code connascences, since the boundary between successful and unsuccessful applications is often drawn in the ability of software to continuously evolve with time and new requirements.

Software is often multidimensional, and change is only one of those facets. But connascence is not only about change, it is about most of the other sides as well. Meilir Page-Jones, the inventor of connascence, mentions that “the understandability and maintainability of object-oriented software - even the value of object orientation itself - rest fundamentally on encapsulation and connascence [4]. For example, by definition software maintainability encloses the notion of correcting misbehaviour, which ultimately is about changing some parts of the system.

Like connascence, refactoring is yet another very important aspect of software, although it should be regarded more as a process of changing code as opposed to measuring the quality of code. Martin Fowler notes about refactoring that “is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour” [9]. There are countless reasons to why refactoring is important for all programmers, but one that stands out more than the others is that the process is aimed at improving the quality of existing code. It can be further thought of as improving the time it takes to fix existing errors or improving the time it takes for new programmers to understand the code and start working with it, amongst many other improvements.

In his book on refactoring, Martin Fowler further adds that the way to proceed in having a successful code transformation is by combining many little steps, less powerful than the overall change, which he coins as “refactorings”. The rationale behind this chain of steps is that as each link “is small, it is less likely to go wrong” [9].

Although seemingly unrelated, the notions of connascence and refactoring fit together, as individual connascence from code can be thought of as one of those links from the refactoring chain that needs to be modified - a small step in improving the system overall quality and flexibility. Connascence can therefore act as the driver for refactoring, or put another way, the checklist to making sure that the software remains flexible to future requirements.

## 1.2 Relevance

Our solution builds on the speed of code metrics, since, after all, connascence is amongst other things a code quality metric. Like all other metrics, connascence is not the perfect solution, but rather a complement to existing tools. However, unlike most other metrics, connascence considers

code from a three-dimensional view: strength, degree, and locality.

All the different sides are meant to help filter out less serious problems, and deliver knowledge about those that actually matter to programmers. When it comes to strength, it should be noted that connascence can be viewed on a scale starting from mild connascences, that often pose little problems to code, to extremely serious problems that if left unresolved will increase code's friction to change. For example, the least serious connascence is connascence of name [4]. We say that multiple components have connascence of name when they must agree on the name of an entity, and this can be the name of a parameter, the name of a method etc. Clearly this is omnipresent across codebases, and it is considered a weak form of connascence since renaming entities is usually an easy task. On the other hand, a stronger connascence is connascence of meaning, which happens when multiple components must agree on the meaning of particular values, like for example the number used to represent the error code of a method. This, in turn, is harder to detect since it requires a semantic analysis, substantially more work than what is required for detection of connascence of name.

The degree property of connascence is meant to differentiate between those entities that are connascent with a lot of other entities as opposed to those that have only a few connascences. Likewise, the locality property is meant to highlight those connascences that pose more threats to the codebase, that is connascences that happen between entities further apart in the system, as opposed to those that are closer within the system.

Thus, unlike most other metrics, connascence does not map the codebase using a single dimension, but rather it takes a more holistic approach, to help inform programmers about those problems that stand out more than the rest. This is one area that most current available tools neglect, since they usually treat all errors, regardless of where they happen, how frequently they happen etc. as being equal, and in most cases programmers are left confused to what is actually the most significant problem that oughts to be fixed. Our work has focused specifically at delivering information that clearly labels the *intensity* of a problem.

In addition to using a code metric, our tool can be also considered a static checker. However, to deliver a better user experience, we narrowed the time between feedback loops of classic static checkers, by providing information either as new code is added or as the programmer inspects already existing components. This is one of the key aspects which differentiates the tool from the other existing solutions, as it means that a user of our tool can potentially be made aware of problems as they appear in code, and even fix them as they arise, rather than running inspections after the whole implementation has been completed.

More often than not, static checkers deliver false positives [15], that is errors that do not actually exist, but which are flagged erroneously due to a limited/poor knowledge of the project. However, thanks to the nature of connascence as code metric, the scope for such errors is limited, which is why our solution improves over most other checkers and delivers substantially less false positives (see section 6.1).

Unlike most other alternatives, our tool provides information specifically aim at what diminishes code's velocity to change, making every suggestion a potential refactoring step, and thus can be regarded as delivering information that definitely impacts all programmers. If some tools focus on detecting specific problems like security issues etc. [16], our approach has been to target the root cause of most problems which decay software and make it less resilient to change.

Therefore, unlike many other existing solutions, our tool delivers information about a whole range of problems, that are prioritised to inform the user what is actually causing a real threat in the codebase. By delivering this experience on the fly, as every new line of code is crafted, the benefits extend beyond running a checker once in a while, and can potentially help avoid creating code problems before implementation is finished.

## 1.3 Objectives

The goals of this project are to enhance existing programming workflows by delivering information to programmers about problems within their codebases that could potentially have negative long-term consequences by increasing the friction to change, and thus leading to decay of software quality. Moreover, within the presented information we aimed to make a clear distinction between different levels of threats, thus providing the user with a clear picture of what needs to be prioritised when attempting to fix such problems.

In addition to highlighting these problems, our aim is to provide user feedback on what can be changed in the codebase to eradicate them, as a means of code suggestions or even automatic transformations that users can apply.

Furthermore, given that change is invariant with software, our focus has also been on designing a solution that is extensible and easily modifiable, such that further enhancements that may or may not build on existing analyses can be appended with ease.

Lastly, one of the optional targets was a constraint on how quickly the tool would need to report errors back to users, but with time it became clearer that we could achieve a solution that worked fast enough to provide real-time detection, that is as every new line of code is added.

## 1.4 Contributions

In this report we present an IntelliJ IDEA plugin, *Connascer*, that provides support for:

- detection of instances of connascence of meaning from user code, a class of problems that may lead to an increased software decay, as detailed in section 4.1
- detection of instances of connascence of position from user code, another class of problems that may negatively impact code's relative velocity to change, as described in 5.1
- detection of connascence locality for both connascence of meaning and connascence of position, to provide knowledge about the impact of detected connascences in the application, as mentioned within both section 4.1 and 5.1
- automated refactoring steps that can be applied to fix detected connascences of meaning, as presented in section 4.2
- automated refactoring steps that can be applied to fix detected connascences of position, as presented in section 5.2
- a visualisation mechanism of connascences that can help users easily identify those with the most impact, and prioritise these for refactoring, detailed throughout sections 4.1 and 5.1, and first described within section 2.4.4
- real-time analysis for detection of both instances of connascence, as described in section 3.1

Additionally, we are proud to mention that the plugin has been accepted as part of the presentations that will be featured within the 21st edition of the *Software Practice Advancement Conference* [10]. More specifically, our solution is scheduled for demonstration on Tuesday, 27th June 2017.

Throughout the project, our mantra has always been that “any fool can write code that a computer can understand. Good programmers write code that humans can understand.” - Martin Fowler [9]. This is our humble attempt to deliver a tool that aims to help programmers write code that humans can understand.

## Chapter 2

# Background

“While software development is immune from almost all physical laws, *entropy* hits us hard”. This is what pragmatic Andy and Dave suggest [17]. Entropy is a measure of molecular disorder within a system. In universe, entropy always tends to a maximum. Unfortunately, in too many software projects, the entropy’s equivalent of code decay also tends to a maximum.

The starting point for code decay is most often a simple mistake, as innocent as a variable left without proper encapsulation, or a method which is left untested. Any mistake which is not fixed leaves an open window for more. Although studied in a completely different context, the *Broken Windows Theory* supports the idea that wrongdoings (or in software terms, code decay) start from very little [18].

The first part of this section presents our research on potential solutions that could help programs improve the quality of their code, and thus prevent a decrease in software’s velocity to change. We start by talking about existing tools that attempt to find problems in code and present them to users. Later on, we talk about a different category of potential solutions for finding code problems, namely code metrics. Our last research topic investigates the Unified Modelling Language (UML) as means of visualising code problems.

The other part of this chapter establishes our proposed solution, and analyses the benefits as well as weakness that come with it.

## 2.1 Static Checkers

The section on static checkers looks at the most used tools that analyse code without actually executing it and aim to detect code problems that may lead to a long-term decrease of software’s velocity to change. We focus on both integrated development environments (IDEs), as more often than not programmers rely on them within their development workflow, as well as tools that either integrate within these IDEs or can be run from the command line.

The goal of this section is to analyse the state-of-the-art in terms of existing solutions that address our established goals as well as to identify their strengths or weaknesses with respect to our requirements.

### 2.1.1 IntelliJ IDEA

As one of the most popular Java IDEs that supports advanced code navigation as well as code refactorings [19], IntelliJ is in itself more than a simple programming environment. Since its inception in January 2001, IntelliJ has grown to feature many detection inspections that analyse

code without compiling it, and delivers potential errors directly to users by highlighting the site of errors.

A screenshot of a code editor showing a Java method definition. The code is: 

```
/** Removes all listeners on this actor. */  
public void clearListeners () {  
    listeners.clear();  
    clear();  
}
```

 A tooltip-style error message is displayed over the `clear()` call, stating "Access can be private more... (%F1)".

Figure 2.1: Access of `clearListener` method can be made private

For example, within figure 2.1, we can observe a potential error suggested by the IDE, namely that the access to method `clearListener` can be private. This is the result of an analysis over the codebase which IntelliJ, like any other IDE, can trigger especially since it builds the graph of relationships between the various components within code.

In the source code presented, the method `clearListeners` is defined as public, however the IDE reports that access to it can be restricted as it is not used in any other part of the system. In this particular case, given the nature of the method, namely to clear registered observers, and that most classes that implement the *Observer* pattern only use this functionality for themselves, this may be indeed a genuine problem to resolve. However, it is up to developers behind such code to decide about it.

Specifically on the topic of inspections, IntelliJ offers a wide variety of options that users can turn on and off, depending on their preference. Part of this list of inspections is provided within figure 2.2.

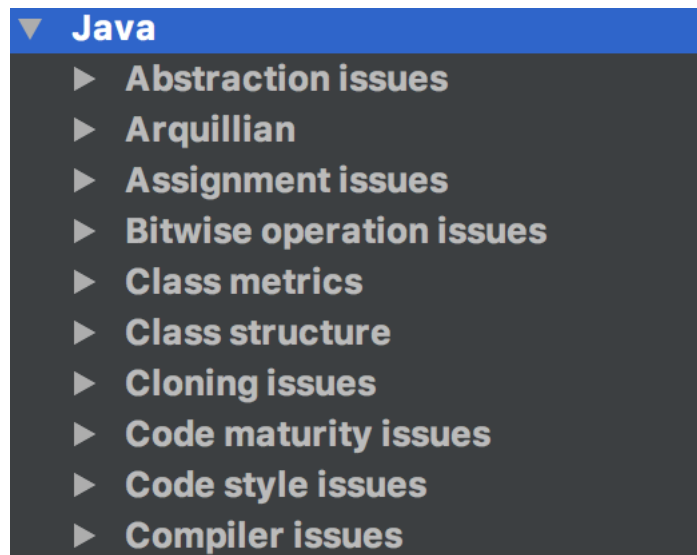


Figure 2.2: Part of IntelliJ's Java suite of inspections

Although a large part of these inspections deal with various easy-to-detect problems, such as reporting unused methods etc., there are some that come closer to our required functionality, like those that aim to detect deep inheritance trees or classes that are highly coupled, meaning that they reference too many other classes, and if change is brought to one of those referenced classes, they may need to change as well.

Another strength of IntelliJ is that in the majority of cases when it reports problems, it also supports refactorings that help programmers solve them. In addition, it also provides an extensible platform to add support for additional refactorings, which is of particular interest given our requirements to provide automatic fixes to help developers remove problems that increase code's friction to change.

However, the most important weakness of the platform is the lack of more advanced inspections, such as those that specifically target code smells which are better indicators of problems that may lead to code's inability to incorporate change easily later on. For example, there is no inspection to indicate that a change in one component may lead to problems in other parts of the system that reference this component.

Additionally, another drawback is that there is no way to differentiate between whether some problems reported by a specific inspection may affect the codebase more than other similar problems reported the same inspection. This is an important missing feature as we would like to provide users with clear priorities in terms of what needs to be refactored first.

### 2.1.2 Checkstyle

Checkstyle is a static analysis tool that can be run either from the command line, or can be integrated within popular IDEs such as IntelliJ IDEA, Eclipse or Netbeans [20].

The goal of the tool is to “help programmers write Java code that adheres to a coding standard”. It further “automates the process of checking Java code to spare humans of this boring (but important) task”.

Besides being conveniently available for virtually all Java developers, Checkstyle provides a catalog of more than 60 detection rules that come installed with the base distribution. Additionally, these rules can be extended as the tool provides a plugin-like environment, where users can add new rules using custom configurations.

Examples of check provides by Checkstyle include those from figures 2.3 and 2.4.

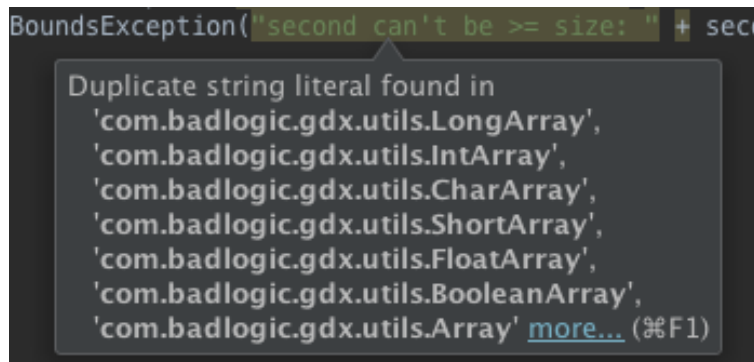


Figure 2.3: Detection of duplicate code

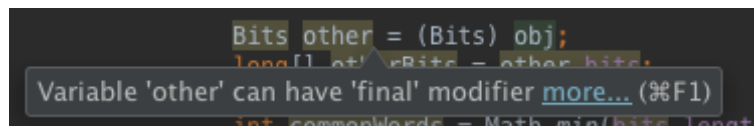


Figure 2.4: Detection of missing *final* qualifier

Although it offers quite a substantial amount of detection rules, these are not concerned with the higher level context problems, like coupling relationships between software components. Instead, all inspections focus on rather local, specialised problems, such as code duplication as presented in figure 2.3 or checking whether access of variables can be made *final* as presented in figure 2.4. Likewise, there is no easy mechanism to help users prioritise what needs to be refactored first, as all problems are treated equally.

Furthermore, some of default inspections also come with refactoring solutions, if used within an IDE environment. For example, a potential fix available for the case of duplicated code presented in figure 2.3 involves refactoring it as a package constant, while in the other case, the *final* modifier is added to the modifier list of variable `other`. However, although custom detection rules can be implemented within the main engine used by Checkstyle, there are limitations in terms of what additional refactorings can be implemented because the platform does not provide a common functionality to implement such additional refactorings. It is up to developers to further seek integration with IDEs.

### 2.1.3 FindBugs

FindBugs is, like Checkstyle, a “program which uses static analysis to look for bugs in Java code”. Similarly, it can be run either from command line, as well from one its integrations with popular IDEs [21].

Unlike Checkstyle, FindBugs does not aim to support detection of problems that arise from disrespecting a coding standard, but it rather targets common pitfalls of the Java programming language. Within figure 2.5, we present the supported general categories for which detection support is provided.

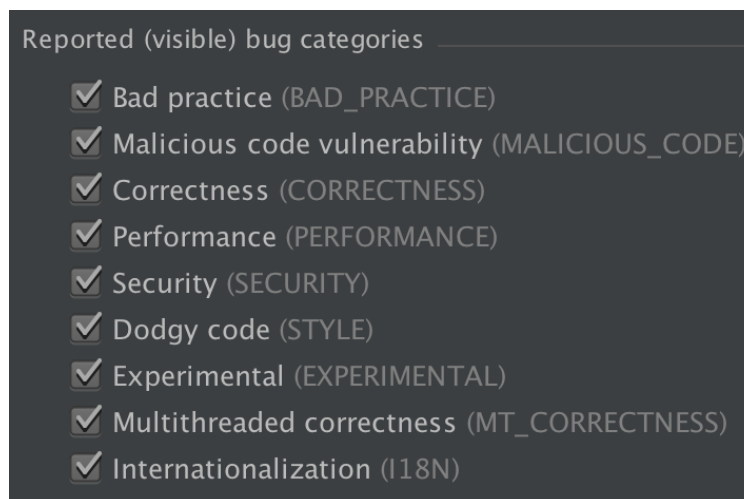


Figure 2.5: Detection categories supported by FindBugs

Notice that this list is similar to that of Checkstyle in respect to the omission of problems that affect higher-level concepts, such as code coupling.

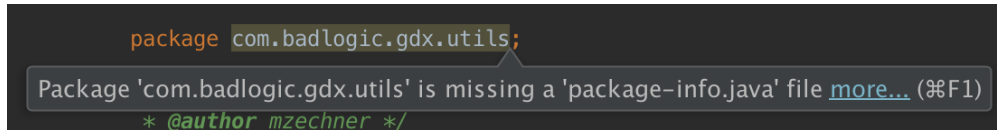


Figure 2.6: Package missing documentation

Within figure 2.6, FindBugs was able to detect the inclusion of a package that does not provide documentation, thus it highlighted this as a potential error because it falls under the case of *bad practice*.

Unlike Checkstyle, there is no easy way to add support for additional detections, which is very surprising taking into account that FindBugs is one of the most used such detection tools by developers.

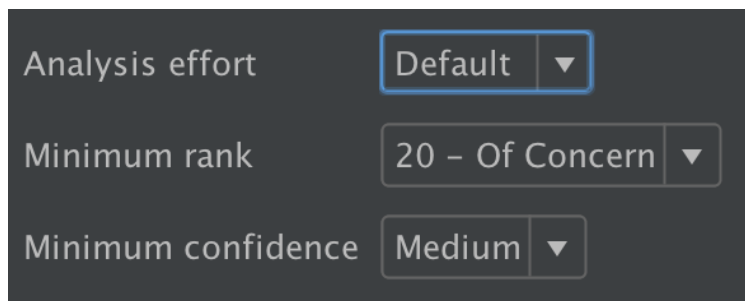


Figure 2.7: FindBugs inspection options

As reflected from figure 2.7, FindBugs supports options that neither IntelliJ nor Checkstyle include by default. One of the most important features is that it can disregard inspections if FindBugs detects their probability of being a false positive under some threshold, which is important with such a tool that aims to detect a lot of problems, as more often than not that raises the possibility of informing users about problems that do not actually exist. Additionally, one of the other very useful features is the ability to turn off detection for problems that fall under certain rank, which FindBugs uses to classify how important a problem is within code. For example, a *NullPointerException* is given a higher rank than a *missing documentation* problem. Unfortunately, although this is a good way for developers to prioritise problems for refactoring, this ignores the possibility of certain problems of the same kind being more important. For instance, perhaps a *missing documentation* is not too important for a relatively unused package of the application, but if the package contains the main API then the same problem becomes much more important.

#### 2.1.4 PMD

PMD is “an extensible cross-language static code analyzer” [22], that provides a relatively similar detection of problems like IntelliJ, Checkstyle and FindBugs.

Unlike the other tools, PMD does not limit itself to Java code checking, but it can also detect problems within other languages such as C, C++, Objective-C, Swift and even vintage languages like Fortran.

Within Java, it claims to have support for detecting problems like:

- possible bugs - empty try/catch statements



- dead code - unused parameters or private methods
- suboptimal code - wasteful String/StringBuilder usage
- overcomplicated expressions - unnecessary if statements, for loops that could be while loops
- duplicate code - copied/pasted code means copied/pasted bugs

It can be used directly from the command line or within one of the popular IDEs, as it supports integration with more than ten of the most popular editors used by developers, including IntelliJ, Eclipse or Netbeans.

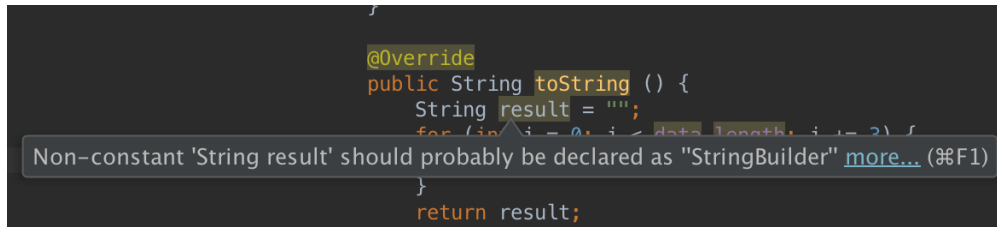


Figure 2.8: PMD Suboptimal code detection

As shown in figure 2.8, PMD is able to capture code problems that are probable bugs, with slightly different detection algorithms than previous mentioned solutions, but which ultimately still ignores problems that concern higher-level code problem concepts.

Although detection rules can be extended by including different *rule sets*, refactorings of such problems cannot be easily provided. Developers must therefore seek to integrate such fixes directly within the IDEs. In addition, PMD also provides fewer options than FindBugs, since it does not offer the possibility to filter out problems with a high probability of being false positives, nor it classifies problems differently in order to offer better guidance for refactoring.

Overall, although it offers additional detection rules, these can be built within other similar products like FindBugs. However, unlike most other tools, PMD can be used with a multitude of languages.

### 2.1.5 Others

In our research we have considered other tools such as Eclipse [28], Coverity [29], JArchitect [30], JTest [31], and Squale [32]. We omitted details about those either because they do not add anything additional over already presented solutions, or in cases like JArchitect, they require a license to use.

## 2.2 Code Metrics

Code Metrics are measures of different aspects of software, that aim to enrich developers with knowledge about their system that may not be easily detected. Often times they are the result of applying formulae using different code measures, such as the number of class references a method uses, or even the number of lines of code of a class.

For example, one of the most well known metrics for detecting software quality is *cyclomatic complexity* [23]. It was developed in 1976 by Thomas McCabe, and it was aimed as a measurement for the complexity of a program, as identified by the number of linearly independent paths within

source code. It was also thought that programs that have a lower complexity also have a better quality.

```
1 public static void main(String [] args) {  
2     if (args.length > 2) // execute program  
3     else // return error  
4 }
```

Code 2.1: Simple main function which processes input arguments

Within code section 2.1, as there is an if-else pair of conditionals, the cyclomatic complexity is two, because each conditional case leads to an independent path of execution. If software contains no condition, the cyclomatic complexity is one, but with any such statement it increases by one.

Mathematically, based on the control flow of the program, the complexity can be defined as a formula in terms of the number of edges and nodes from this graph, as well as the number of connected components. So if we assume  $E$  to be the number of edges,  $N$  to be the number of nodes, and  $P$  to be the number of connected components, the cyclomatic complexity is equal to:  $E - N + 2P$ .

Although there is some utility attributed to this complexity measure [24], some researchers have proven to be this metric as predictive of the number of defects within code as the metric of the number of lines of code [25]. Furthermore, since the metric is based on the computation over the whole control flow graph, it has the downside that for large software it may take a long time to compute. Given the necessary computation underlying the complexity formula, real-time analysis may only be provided for toy programs.

Aside from a longer time that is required by some metrics to be compute, most metrics have the advantage to deliver information to users about the probability of underlying source code to contain real errors that may lead to software decay. Those are different errors to uninitialised variables which some static analysis tools report, and focus more on the ability of a software component to be cohesive by itself. For example, the *Tight Class Cohesion* metric computes the relative number of method pairs of a class that access in common at least one attribute of the measured class [3].

Class metrics surely come close to some of our requirements to identify possible errors that may lead to an increased friction to change within code, but one of their biggest disadvantage is that most of the time they map source code to a number using a transformation function, thus losing important information about the context where such metrics are computed.

## 2.3 Unified Modelling Language (UML)

UML is a modelling language, invented by Grady Booch, Ivar Jacobson and James Rumbaugh in late 1990s [26]. The language hides away code details such as algorithms, or specific code constructs, to put emphasis on the overall system design. For this reason, it has been the standard of system design visualisation used in the field of software engineering.

One of the reasons why we present UML is because of its power to present users with abstractions that can help understand the overall design and potentially detect problems that may arise from it.

Within UML diagrams individuals classes are represented by boxes, and their properties such as variables and methods are captured within boxes. In addition, relations between classes such as inheritance or ER-relationships - *is a*, *has a* [33] - as well as interactions between objects within systems are represented by arrows that link various classes.

As such diagrams hide away details from code, it is easy to visualise the resulting system. However, within large systems such visualisations may become incomprehensible, due to the large possible number of interactions. Within such systems we can identify potential problems by targeting cluster of components that have many links between them, as well as components that have long relationships within the system. For example, we expect classes within the same package to have a reasonable amount of communication between them, and likewise, we do not expect to see too many relationships to components outside the package.

Although the language is very powerful at creating such diagrams, it is less so powerful at conveying information about underlying components. This represents of the biggest weakness in terms of identifying problems that lay within the code of such components.

## 2.4 Proposed Solution

Connascence is a software quality metric, that unlike most other code metrics regards code through a three-dimensional approach. The three facets include strength, locality and degree [4].

In this section we introduce connascence through code examples, present the rationale behind using connascence as a driver for finding possible errors that affect software's long-term velocity to change, as well analyse the three dimensions of connascence and how they impact our project requirements.

In addition, we specify how our solution meets the real-time analysis and automatic refactorings requirements, and conclude by putting in balance strengths and weaknesses of our approach.

### 2.4.1 Connascence

Connascence is a code metric and taxonomy for different types of coupling invented by Meilir Page-Jones [4]. Like all metrics, connascence is not a perfect measure, but unlike most other metrics, connascence does not map source code to a number, but rather regards it through different facets.

The origin of the term connascence is from Latin, and stands for having been born together. In software terms, that refers to components which share a coupling relationship such that, if a change is brought to one, the other components that are part of the relationship must change as well in order to preserve overall system correctness. One of the basic examples of connascence is when a method defined in a class changes its name, the other parts of the system that use method must follow the change.

```
1 public class Database() {
2     public void storeEntry(Object entry);
3 }
4
5 public class WebCrawler() {
6     Database mainStorage; // initialised inside constructor
7     ...
8     public void cacheWebsite() {
9         Website website = getLastVisitedWebsite();
10        mainStorage.storeEntry(website);
11    }
12 }
```

Code 2.2: WebCrawler that stores visited websites in a database

If now we suppose that after some refactoring, the `Database` changes its `storeEntry` to `archiveEntry`, then if `WebCrawler` class does not implement this change as well, the software ceases to work.

```

1 public class Database() {
2     public void archiveEntry(Object entry);
3 }
4
5 public class WebCrawler() {
6     Database mainStorage; // initialised in constructor
7
8     public void cacheWebsite() {
9         Website website = getLastVisitedWebsite();
10        mainStorage.storeEntry(website);
11    }
12 }

```

Code 2.3: Code from WebCrawler no longer compiles

Such a problem is often minor within a system, as renaming methods is usually a relative easy task, and the connascence *strength* property reflects it. Part of the *strength* dimension is a list of different types of connascences that differ in terms of how hard it is to discover, and to refactor/correct them.

One of the main arguments why we decided to use connascence as a driver for determining code problems that may affect code's friction to change is because, fundamentally, connascence is about coupling relationships between components that need to change together in order to preserve connascence. Our aim is to therefore identify such relationships and help users remove them, such that when further change is brought to the system, only a very small amount of components need to change.

Unlike other solutions which identify problems but do not quantify how much impact they have on the underlying software, connascence prioritises the most severe problems over those that affect the system less. This is achieved through the different sides of it, strength, locality and degree, and in their respective sections we present the mechanism for differentiating between such problems.

Finally, problems identified through connascences can usually be fixed by following a specific algorithm. Taken into account that our aim was to deliver automatic fixes to detected problems, this provides us with yet another opportunity to help deliver on this goal.

## 2.4.2 Connascence Strength

The *strength* property gives us a way to quantify whether a type of connascence is “stronger” than another.

There are nine different types of connascences [4], which start from the connascence of name, and go up on the scale of strength to the connascence of identity.

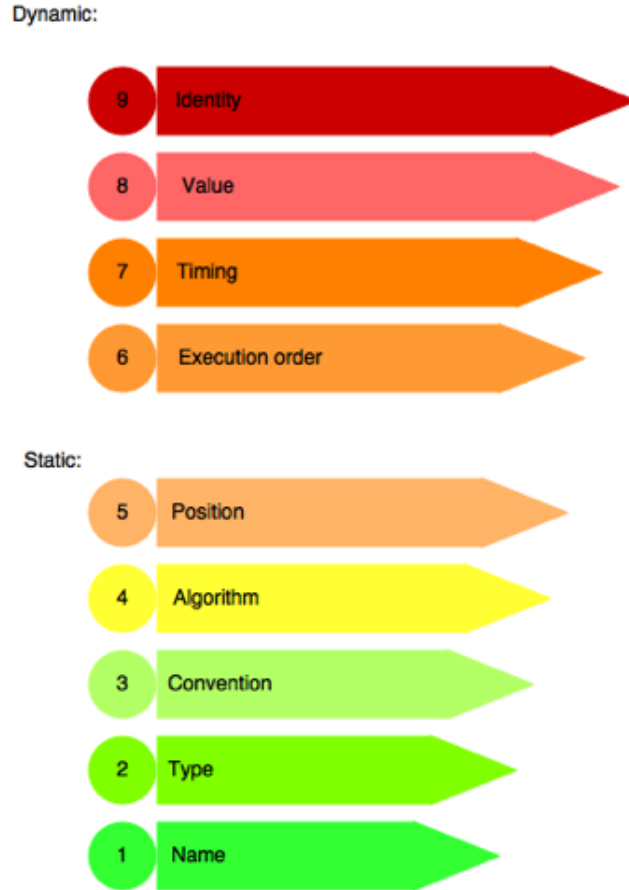


Figure 2.9: Different types of connascences in ascending order of strength

As we can see from the scale provided in figure 2.9, each connascence is assigned a colour. The meaning behind it is that stronger connascences, those higher up on the scale, pose more threat to the software and its ability to maintain a low friction to change than those lower down on the same scale. This provides the ability to prioritise problems within a list of connascences.

Another important aspect of this scale is that it depicts how connascences can be divided within *static* and *dynamic* categories. Static connascences are those which are thought to be detectable directly in the source code, while in the case of dynamic connascences, they are thought to be detectable within running software [4].

In the time available to the implementation of our project we have mainly focused on detection of the connascences of convention/meaning and position, since those were the most powerful ones for which detection was not implemented, thus within this section we present those types of connascences. In addition, we present the rest of other static connascences as we refer to them within the refactoring sections of both connascence of meaning and connascence of position.

#### 2.4.2.1 Connascence of Name

Connascence of name is when multiple components must agree on the name of an entity [4]. Our previous example from code section 2.2 is an instance of a connascence of name. In the code section, both `WebCrawler` and `Database` must agree on the name of the method that stores information in the storage, and as demonstrated previously, if the name of this method changes within `Database`,

the `WebCrawler` must also implement the same change.

### 2.4.2.2 Connascence of Type

Connascence of type is when multiple components must agree on the type of an entity [4]. Using the same example as before, but this time leaving the name of the `Database` storage method unchanged, we only modify the type of its sole parameter.

```
1 public class Database() {
2     public void storeEntry(Entry entry); // previously entry was of type Object
3 }
4
5 public class WebCrawler() {
6     Database mainStorage; // initialised in constructor
7
8     public void cacheWebsite() {
9         Website website = getLastVisitedWebsite();
10        mainStorage.storeEntry(website);
11    }
12 }
```

Code 2.4: `WebCrawler` does not call `storeEntry` method with the correct argument type

As before, unless `WebCrawler` changes the type of the argument it passes to `storeEntry`, the functionality is broken. Once again, this is a relatively easy to fix problem, and as a result it is only listed as the second type of connascence on the strength scale.

It should be clear from the previous two examples that software cannot be written using at least some form of connascence, either of name or of type. That is why on the scale of connascence strength they are ranked as the lowest connascences, and as presented in figure 2.9, they are coloured with green to signify that such instances of connascence pose the least problems in code.

### 2.4.2.3 Connascence of Meaning/Convention

This is the first type of connascence that starts to have a higher impact on code. Components have connascence of meaning if they must agree on the meaning/convention of particular values. For example:

```
1 public class Account {
2     public int number; // negative for business, positive for regular customers
3 }
```

Code 2.5: Users of `Account` must respect the convention set for business and regular customers

Within the `Account` class, its variable `number` has the convention that positive values represent regular customer, while negative ones represent business customers. With such a connascence, is easy to imagine that it can quickly propagate within the system, because whenever users must to differentiate between the type of an account, they will use the convention that negative numbers stand for business customers, and positive number for regular customers.

```

1 public class Bank {
2     public void transfer(Account to) {
3         if (to.number < 0) // business customers
4             else // negative customers
5     }
6 }

```

Code 2.6: Meaning of negative and positive numbers of `Account` is further propagated within system

#### 2.4.2.4 Connascence of Algorithm

Connascence of algorithm is when multiple components must agree on a particular algorithm [4]. For example, if data sent by a server is encoded using a specific cryptographic algorithm, it must be decoded by the client that receives it with the exact same algorithm.

```

1 public class Server {
2     public void sendData(Data data) {
3         socketTransmit(md5(data));
4     }
5 }
6
7 public class Client {
8     public void receiveData() {
9         Data data = md5(socketReceive());
10    }
11 }

```

Code 2.7: Server and client know they must use md5 algorithm for encryption/decryption of data

#### 2.4.2.5 Connascence of Position

Connascence of position is the most powerful of static connascences and happens when multiple components must agree on the particular ordering of values [4]. To exemplify, please consider the following code:

```

1 public int[] getUserDetails() {
2     int[] details = new int[2];
3     details[0] = getUserId();
4     details[1] = getUserAccessLevel();
5     return details;
6 }

```

Code 2.8: Method which retrieves id and access level of a particular user

Based on the return type of the `getUserDetails`, somewhere else in the system there will be usages of it that specifically target either the id or the access level by indexing within the array:

```

1 public void executeScript() {
2     int[] user = getUserDetails();
3     if (userHasExecutePermissions(user[1])) // execute script
4     else // register attempt from user id, user[0]
5 }

```

Code 2.9: Method that targets specific values within the retrieved array from `getUserDetails`

### 2.4.3 Connascence Locality

Connascence location is primarily concerned with how close or far connascent components are within the system, and we often classify it within one of three different options: within class, within package and outside package. Different localities help us prioritise connascences a system, such that we treat those that cross package boundaries, that is those that are the furthest apart in the system, as being the most damaging for software. We then treat those connascences that happen within classes of the same package as more powerful than those that happen within class.

Building on a similar example to that shown previously within the *connascence of meaning* section, consider the following code snippet:

```
1 public class AccountNumber {
2     ...
3     public int accountNumber;
4     ...
5
6     public boolean isBusinessAccount() {
7         return accountNumber > 0;
8     }
9     ...
```

As within the `isBusinessAccount` method we have a comparison between `accountNumber` and the literal 0, this reveals an underlying convention used for `accountNumber`, thus this is an example of a connascence of meaning. Since this connascence happens within the `isBusinessAccount` method, which is inside `AccountNumber` class, and as `accountNumber` is an instance variable of the same class, we classify its locality as within class, that is the weakest type of a possible connascence.

Suppose now that we within the same application, there is another component that uses the `AccountNumber` class.

```
1 public class Transaction {
2     ...
3     public void transferCash(AccountNumber transaction) {
4         if (transaction.accountNumber > 0) { // use quick transfer, business accounts
5             ...
6         } else { // use normal transfer, regular accounts
7             ...
8         }
9     }
10    ...
11 }
```

Code 2.10: Another usage of `AccountNumber` within the same system

As an additional assumption, consider that `Transaction` class is in the same package with the `AccountNumber` class. In this case, as we have already discovered a connascence of meaning between these classes, namely through the usage of `accountNumber` within the comparisons made inside the `transferCash` method, we classify its locality to within package, a more serious type of connascence than of that within class.

Lastly, if we consider `Transaction` to be in a different package to `AccountNumber`, the connascence of meaning would be then classified as having a locality of type outside package, the most serious such offence.



Connascence Locality is extremely important in achieving our goals of allowing users to differentiate between serious problems and those that are less serious, since, for example, users can prioritise those most important violations within their code for refactoring.

Additionally, code cannot exist without connascence, since any coupling relationship can be classified as connascence. In his book on the topic of connascence [4], Meillier Page-Jones mentions how connascences inside methods are useful since this can be a good indicator of reusing other existing components like helper methods, how those within a class are alright if care is applied in the class design, and how connascences that start crossing boundaries, namely class boundaries and package boundaries tend to have a very powerful negative effect on code's future velocity to change, as a "change in a component must follow a change in its connascent components to preserve overall correctness".

Therefore, to deliver an accurate information to users about the severity of problems within their codebases, we have decided to embed connascence locality detection algorithms in our plugin for both connascences of meaning, as well as connascence of position.

#### 2.4.4 Connascence Visualisation

One of the key components of our analysis is the visual feedback we provide to users, since this is our way to signify the site of a connascence. Our choice to convey the visual feedback is by highlighting the code affected by connascence problems, so that users can learn immediately about such problems when inspecting a file.

To differentiate between various connascence locality levels, we decided to provide distinct colours for each locality type, such that weak connascences, those that have a locality within class merely get highlighted by means of underlying code (`_`), those that have a more serious level of locality, within package, are highlighted by using a warning-like error yellow colour (`■`), and the most serious offences are highlighted using using a similar red that resembles that used by IDEs to highlight compilation errors (`■`). The scheme has been chosen to help users directly visualise the most serious offences, while also providing easy-to-notice feedback about those that are less serious.

To help visualise of our colouring scheme, please consider the above `AccountNumber` and `Transaction` classes once again, that are within the same `account` package, and an additional class `BankTransaction` which is in a complete different package, `banktransactions`:

```
1 package account;
2
3 public class AccountNumber {
4     ...
5     int accountNumber;
6     ...
7
8     public boolean isBusinessAccount() {
9         return accountNumber > 0;
10    }
11    ...
12 }
```

Code 2.11: Within class connascence of meaning inside `isBusinessAccount` method

```

1 package account;
2
3 public class Transaction {
4     public void transferCash(AccountNumber transaction) {
5         if ( transaction.accountNumber > 0 ) { // use quick transfer, business customers
6             ...
7         } else { // use normal transfer, regular customers
8             ...
9         }
10    }
11 }

```

Code 2.12: Within package connascence of meaning between **Transaction** and **AccountNumber**

```

1 package banktransactions;
2
3 public class BankTransaction {
4     ...
5     public void externalTransaction(AccountNumber transactionOrigin) {
6         if ( transactionOrigin.accountNumber > 0 ) {
7             ...
8         }
9     }
10    ...
11 }

```

Code 2.13: Outside package connascence of meaning between **BankTransaction** and **AccountNumber**

Additionally, please note that in most snippets where identification of connascences is presented in sections 4.1 and 5.1, we may refer from highlighting code in order to put more emphasis on presenting why the underlying problem should be considered a connascence in the first place, thus leaving the visual feedback out to enhance code readability. Furthermore, locality can be determined only when both the source and the destination - where connascence actually takes place - of a connascence are known, and as our provided examples are brief, they do not provide all necessary details that would allow to determine the locality, such as enclosing packages.

### 2.4.5 Connascence Degree

The last dimension of a connascence is the degree. Essentially, this is the number of connascent components that exist within the application and have the same origin. For example, within code sections 2.11, 2.12, and 2.13 the origin of the connascence is always within the **AccountNumber** class, and since these are three distinct instances, the degree of this connascence is three. This becomes really useful in helping further prioritise connascences, as those that have the same strength and locality are ultimately differentiated by degree, that is by how much impact they really have within the system.

In order to compute the degree property, one must search throughout the entire source code to find possible usages of a specific connascence, and as our requirements involved a real-time analysis component this property could not have been fitted, at least in the current iteration, within our final detection algorithms.

## 2.4.6 Real-time analysis and Refactorings

Given the limited time available for implementation allocated to this project, we decided to use existing solutions for hosting our analysis and refactorings. One product that stood out more than the rest was IntelliJ IDEA, since as presented within section 2.1.1, this is an IDE that already provides support for code highlighting, as well as a rich SDK that programmers can use to develop their own inspections and refactorings. As presented within the implementation section, our work has been greatly accelerated as a result of having to focus specifically on implementing tasks such as connascence detection, and using IntelliJ's facilities for less-so-important tasks such as source code parsing.

Additionally, to achieve the real-time requirement within our analysis we employed an architecture that distributes tasks among available threads to maximise the potential for parallel work. Full details about how this is achieved in our solution are laid down within section 3.1.

## 2.4.7 Final remarks

As we have already mentioned, our proposed solution is based on a code metric, and like all code metrics, this is not the perfect measure that solves all problems. However, it should be noted that since connascence targets mainly components that need to change together, it delivers a better knowledge about underlying problems that may affect code's velocity to change than most other alternatives.

Perhaps one of the downsides of connascence is that it does not directly target inheritance relationships that so very often lead to an increasing friction to change within code, however, as mentioned within section 2.1.1, there are already implemented solutions that attempt to target such problems.

In addition, a benefit on relying on connascence for detection of such problems is that it enables us to deliver a real-time analysis that may help users prevent errors as they are added within the application, as well as possible refactorings to help users correct problems reported by our analysis within their codebase.

Another weakness of connascence is that it should be mainly used for problems that cross package boundary, or at least class boundaries. Thus, connascence problems detected within the same class may not be always relevant. A complementary metric to help with this weakness is that of islands of cohesion. Ideally, connascence should be considered for problems that, as mentioned, cross class/package boundary, while inside of a class, the more relevant metric is that of islands of cohesion.

Finally, delivering our solution as a plugin for the IntelliJ platform has provided us with facilities for tasks such as code parsing, that ended up allowing us to deliver a better overall product by spending more time on more important aspects of our project, such as the underlying detection algorithms. However, it also restricted us from providing our solution to users who do not rely on IntelliJ as part of their development workflow.

## Chapter 3

# Implementation Design

*Connascer* is developed as an IntelliJ IDEA plugin that offers real-time detection of connascences of meaning and connascences of position. In addition, it suggests solutions that developers can trigger to automate the refactoring process and remove underlying connascences.

To illustrate the overall design behind *Connascer* we start by presenting how we are able to detect connascences in code. We then discuss the design behind delivering visual feedback to users about detected problems. Finally, we illustrate the architecture behind code refactorings and talk how individual detection algorithms fit within the overall design.

### 3.1 Problem detection

One of the most important goals of our project was to achieve detection of problems that could negatively impact code's friction to change, namely connascences of meaning and connascences of position, as we have discussed in section 1.3.

#### 3.1.1 Source code parsing

To detect problems in source code, one must first develop a semantical understanding of it, that is, the meaning behind code statements. For example, within the code section 3.1, one must detect that `foo` is a method of the current object, which is invoked with `bar` and `biz` as arguments.

```
1 this.foo(bar, biz);
```

Code 3.1: Possible method call statement in Java

This is exactly what compilers do all the time as part of making sure that code is valid [34]. They take source code as input, and generate an *Abstract Syntax Tree* from it, which is a tree-like data structure that supports a fast traversal of code elements. In addition, since static analysis tools do not compile the actual code, this mechanism is usually replicated within such tools.

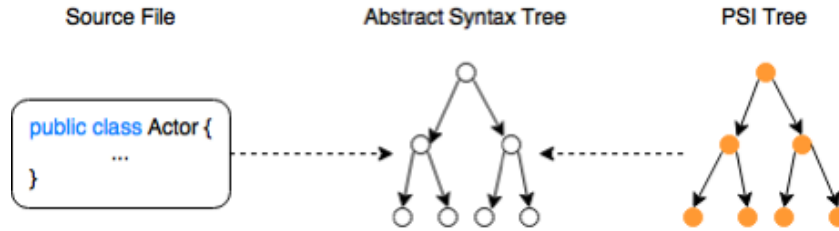


Figure 3.1: From source to IntelliJ's PSI Trees

IntelliJ provides such a similar parsing mechanism, and as presented in figure 3.1, it further enhances the abstract syntax tree functionality by wrapping each node within a *Programming Structure Interface (PSI)*, thus delivering additional features like in-node key-value data storage, as well as additional methods to determine the Java file where the element is located etc. For example, the code section 3.2 is transformed by IntelliJ as depicted in figure 3.2. Please note that all individual nodes like `PsiFile`, `PsiPackageStatement` etc. inherit from the `PsiElement` object, much like any object in Java inherits from `Object` class. This is to require common methods such as node data storage `get` and `put` to be implemented by all subtypes.

```

1 package webservice;
2
3 public class WebServer {}

```

Code 3.2: `WebServer` class within `WebServer.java` file

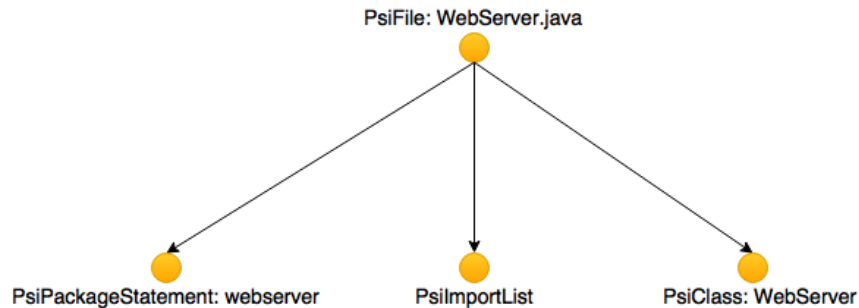


Figure 3.2: Psi Tree of `WebServer`

Based on the PSI infrastructure provided by IntelliJ, we were able to use node visitors such as shown in code section 3.3, to traverse those nodes of interest from a PSI tree.

```

1 public abstract JavaElementVisitor extends PsiElementVisitor {
2     public void visitClass(PsiClass aClass) {...}
3     public void visitPackageStatement(PsiPackageStatement statement) {...}
4 }

```

Code 3.3: Visitor that traverses PSI nodes associated with Java elements

### 3.1.2 Detection algorithms

Based on the understanding built by traversing the source code as discussed previously, we run different detection algorithms, which we may refer to as inspections, that check whether specific code smell patterns exist in code. For example, as discussed within section 4.1 and 5.1, we built rules to detect connascences of meaning and position respectively. In most cases, we feed the source code of currently opened files in the IDE editor as input sources to the detection algorithms. However, this is not a limitation since, if required, each detection can request any additional file it may find useful.

Due to the different requirements of each detection algorithm, as well as to meet our goal of extensible design, each detection algorithm is free to interpret the source code in whatever way it finds meaningful. For instance, an inspection may only require to find problems within class methods, and therefore its source file visitor would be focused specifically on traversing class methods only, and skipping other elements. Likewise, an inspection may require to know the broader context of class methods, perhaps the enclosing class and its other members like class fields etc., in which case the source code visitor would be required to do more work.

Furthermore, with time we realised that the functionality provided by PSI nodes was not enough for detection of the patterns we were interested. For example, although within the PSI interface there is a specific `PsiLiteralExpression` class, to check whether a PSI node is a literal it is not sufficient to check that it is an instance of a `PsiLiteralExpression`, because signed literals, such as negative numbers like -1 are, in fact, `PsiPrefixExpressions` that take an operand of type `PsiLiteralExpression` and an operation sign of type `PsiJavaToken`. Like this small example there were many others that ended up being duplicated in various inspections. Therefore, in order to address this problem, we set up a utility library that provided those `PsiElement` interactions that were not part of the functionality provided by IntelliJ, as well as included common functionality required by inspections such as the ability to request additional source code files. We refer to this library as the project SDK.

Although initially both detections of connascence of meaning and connascence of position performed code analysis within entire Java classes, with time we realised that we could analyse scopes as small as individual methods and still be able to detect all the required information about whether such methods contained patterns that were indicators of connascences.

This discovery was essential in facilitating our real-time analysis for a few reasons. First of all, each inspection can be distributed over multiple threads. Since each method is in itself a scope big enough for detection of various patterns, each inspection distributes the collection of methods of a class to different threads, with each thread running the algorithm behind inspection on all provided methods. This is illustrated in figure 3.3.

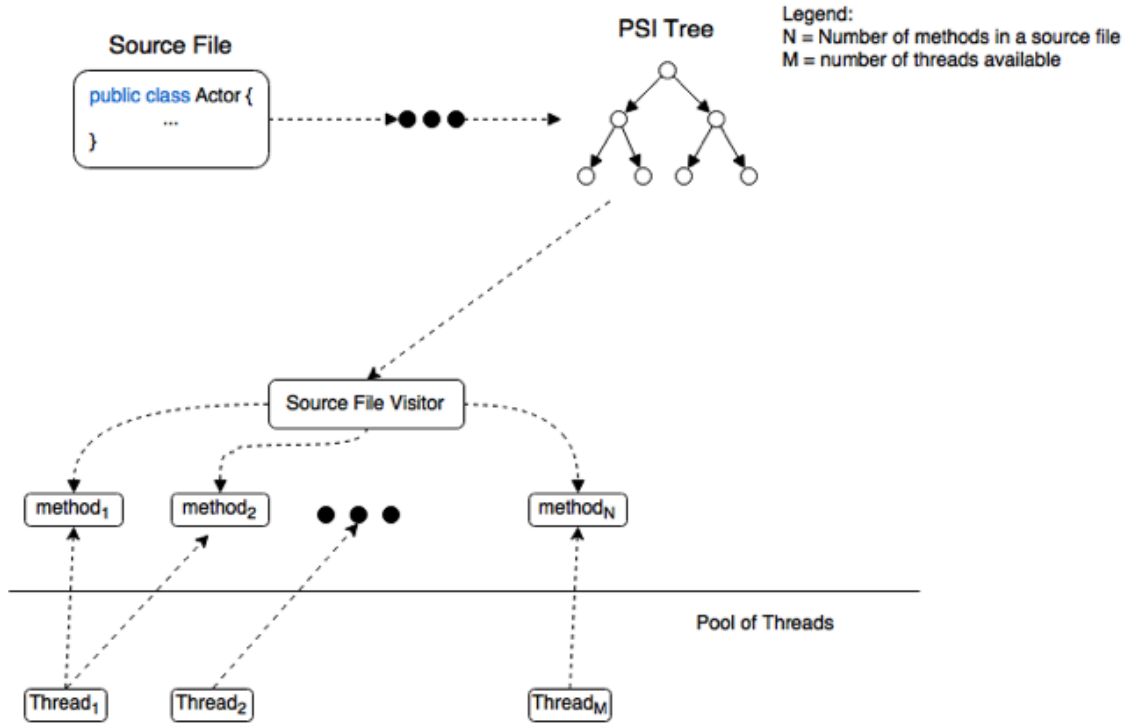


Figure 3.3: Detection algorithms distribute their workload to multiple threads

Additionally, another benefit of relying on inspection of methods is that if a user changes code within a file, the change is most likely made within the scope of a class method. This allows us to disregard/skip inspection of methods which have not changed, and only reanalyse the method were changes were made. For example, in a class with 15 methods, in most cases when changes are made we can ignore 14 out of those methods, and only reanalyse the one that was affected by changes. Additionally, to make sure we do not miss problems that may be an indirect result of changes brought to specific methods, we also run a whole class analysis after a certain time threshold. As previously discussed, this can be completed by parallelising the workload, such that the overall performance overhead added is minimal.

Therefore, given the benefits of distributing detection workload among various threads, as well as being able to ignore a large part of class most of the time, we have been able to run our connascence inspections after each code change, thus providing a real-time analysis. In most cases, as mentioned previously, we only analyse the method where additions or modifications are made, but whenever we inspect a class for the first time, or after a certain time threshold we perform a full class analysis. We provide this functionality as part of our project SDK, but future inspections are able to decide whether a method-based analysis is sufficient, or conversely, if they require knowledge from a broader scope. Even if an inspection requires a broader scope analysis, the project SDK still has support for that, but it may not guarantee the real-time analysis property in that case.

## 3.2 Problem visualisation

To visualise problems reported by existing inspections, as well as those of future inspections, we use a common problem interface, `ConnascenceProblem`, which inspections use to report errors to a problem registration engine. As depicted in figure 3.4, after each inspection is finished, a list

of `ConnascenceProblems` is generated. Each inspection sends the generated list to the central `ProblemRegistrationEngine`, which takes care not only about displaying such problems in the IDE editor, but also to filter out problems such as those that users have blacklisted. For example, as each connascence has an associated locality property (section 2.4.3), users can opt to ignore connascences with a relatively weak locality.

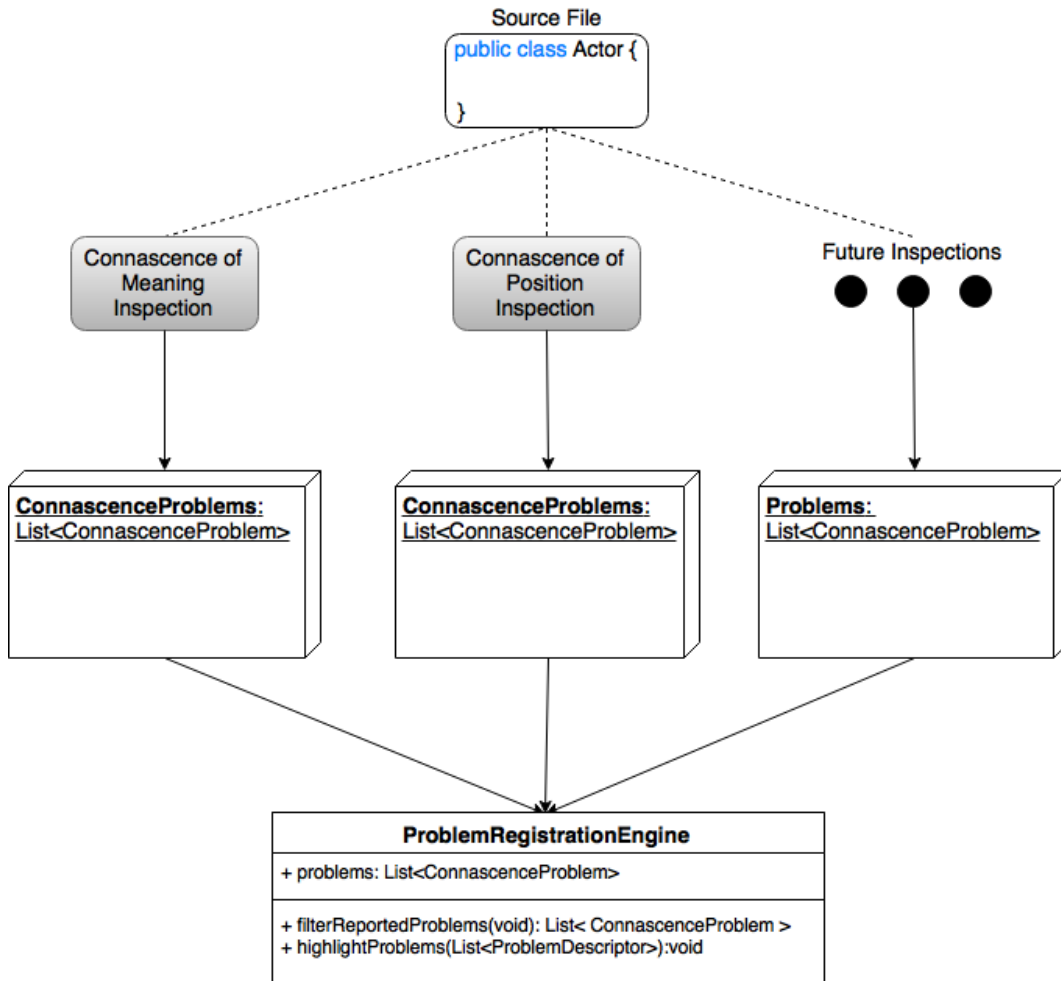


Figure 3.4: Overview of highlighting process

Therefore, in such case, the `ProblemRegistrationEngine` will disregard reported connascences with a weak locality, and only highlight connascences with other types of locality.

```

public V removeKey (K key) {
    Object[] keys = this.keys;
    if (key == null) {
        for (int i = 0, n = size; i < n; i++) {
            [Connascence of Meaning] Comparison against null has an implicit meaning more... (%F1)
            V value = values[i];
        }
    }
}
  
```

Figure 3.5: Example of a connascence of meaning highlighting

As an example, figure 3.5 presents highlighting of a connascence of meaning. The red colour



associated with it stands for a connascence locality of outside package, as discussed in section 2.4.4.

### 3.3 Problem refactoring

Problem refactoring is our mechanism to help users not only learn about problems within their code, but also to aid them in fixing such problems. As refactorings are dependent on the type of problem, each detection algorithm is responsible for providing such fixes. For example, within code section 3.4, the comparison against `null` at line number 3 represents a connascence of meaning, as described in section 2.4.2.

```
1 public void deleteAccount() {
2     Account currentAccount = getCurrentAccount();
3     if (currentAccount == null) {
4         ..
5     } else {
6         ..
7     }
8 }
```

Code 3.4: An example of connascence of meaning

In this case, since the convention used for the comparison is to verify that the `currentAccount` value exists, we can provide a solution that makes this check explicit, as demonstrated within code section 3.5.

```
1 public void deleteAccount() {
2     Account currentAccount = getCurrentAccount();
3     if (Optional.ofNullable(currentAccount).isPresent()) {
4         ..
5     } else {
6         ..
7     }
8 }
```

Code 3.5: Refactoring applied to remove the connascence of meaning

Since the nature of refactorings is to change the underlying problem, and taking into account that inspections such as those provided by *Connascer* run so frequently, an essential aspect is that before a refactoring is applied, it must acquire the read and write lock associated with the source file of the problem. This is to avoid situations where the underlying elements checked by an inspection become invalid as a result of a refactoring. In such cases, PSI nodes checked by inspection become `null`, and thus further accesses to information about such nodes within inspections may result in errors.

The same `ProblemRegistrationEngine` also takes care of making the refactoring of a problem available to users. As shown in figure 3.6, problems reported by inspections are of type `ConnascenceProblem`, which provides access to a `refactorings` field, as generated by their respective inspections. Thus, any new inspection must also provide a list of possible refactorings for the reported problems.

Refactorings also share a common interface, namely `LocalQuickFix`, that enforces developers to provide support for a few things, such as refactoring name, description etc., as well as to provide an implementation for the `applyFix` method, which gets triggered by IntelliJ whenever users request to apply a refactoring.

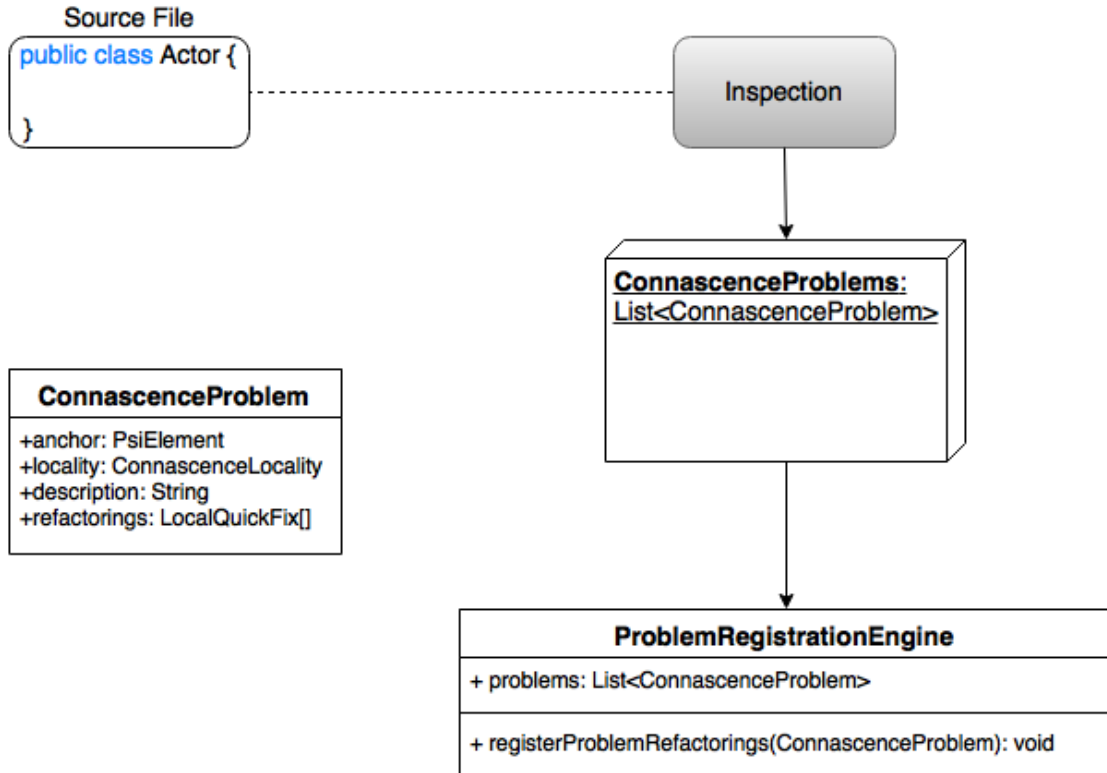


Figure 3.6: Problem refactoring is handled by `ProblemRegistrationEngine`

In the next chapters we discuss specific inspections like that of connascence of meaning and connascence of position. We present which detection patterns they look for in code, as well as mention about which refactorings they use to solve detected problems. Such inspections follow the design discussed within section 3.1.2, and therefore fit within the whole project design as presented in this chapter.

Additionally, as it is beyond the scope and limit of the project report, we do not go into details how each inspection pattern is implemented in code, but rather limit to code examples where such patterns are found, as well as provide rough details about how the locality property (section 2.4.3) is computed in each case. The source code of this plugin is also going to be made available as part of the final project requirements, and the interested reader should request a copy through the Department of Computing at Imperial College.

## Chapter 4

# Connascence of Meaning

Software components have connascence of meaning if they must agree on the meaning/convention of particular values [4]. For example, suppose we have the following `AccountNumber` class:

```
1 public class AccountNumber {
2     ...
3     public int accountNumber; // positive for business customers, negative for regular
4     ...
5 }
```

Code 4.1: Possible implementation of an `AccountNumber` class within a larger application

It is therefore expected that somewhere else in the system, a code snippet like that from code 4.2 exists.

```
1 public class Transaction {
2     ...
3     public void transferCash(AccountNumber transaction) {
4         if (transaction.accountNumber > 0) { // use quick transfer, business customers
5             ...
6         } else { // use normal transfer, regular customers
7             ...
8         }
9     }
10    ...
11 }
```

Code 4.2: Possible usage of `AccountNumber`

It is clear that whenever we need to know the type of an `AccountNumber`, we are going to use conditionals as we did above. A problem that arises with such code is the lack of clarity inherent with such statements. If it had not been for the comment within the `AccountNumber` class, we could have easily mistaken regular customers for business customers or vice-versa. Furthermore, if the application was updated to support another type of customers and the comment remained unchanged, that would have left the room wide open for potential errors.

The topic of comments within code is old and controversial. One of the best summaries we could find comes from Robert Martin, who says that “nothing can be quite so helpful as a well-placed comment. Nothing can clutter up a module more than frivolous dogmatic comments. Nothing can be quite so damaging as an old cruffy comment that propagates lies and misinformation”

[35]. Although there is an indication to what positive and negative account numbers mean, albeit hidden within the `AccountNumber` class itself, this is not clear in any other part of the system where `accountNumber` checks are made.

The work we present in this chapter is on the topic of finding such connascences and taking steps to address them, by means of automatic refactorings, because we strongly believe that “comments do no make up for bad code” [35], and that the expressivity and functionality of software is enhanced without such connascences.

## 4.1 Identification

We have already seen one possible instance of connascence of meaning, however, in reality there are many more patterns where such problems can arise. In this section we present our identification algorithms which cover most frequently met patterns in codebases, along with rough guidelines about how we compute the connascence locality in each case. Furthermore, where appropriate, we use pictures captured directly from our plugin interface, and in other cases we provide  $\text{\LaTeX}$  code snippets that mimic its interface, for ease of reading. Please also note that if code is highlighted, it is with respect to guidelines set in section 2.4.4.

### 4.1.1 Binary Expressions

One of the most frequent pattern for instances of connascence of meaning is found within binary expression, that is expressions that use a binary operator such as “==” or “>” between a literal and a non-literal, as observed in the introduction of this chapter.

We consider expressions such as raw instances of `String` like “Hello World!”, raw usages of numbers like `-1`, `+42`, `0`, the `null` keyword, as well as boolean operators like `true` and `false` - depending on the context - all being instances of literals. Thus one of the requirements for detecting a connascence of meaning within binary expression is to identify one of its operands as being a literal.

On the other hand, non-literal expressions are more complex, since they can different shapes. We present the three different categories that our plugin considers. Additionally, since connascence locality must be computed differently for each category, we also add details about how our algorithms compute it in each case. Please note that all these different cases build on top of IntelliJ’s `PSI` interface [36], thus each case is adjusted to a code equivalent. For example, reference expressions are the code equivalent of `PsiReferenceExpressions`.

#### 4.1.1.1 Reference Expressions

A reference expression is one where the underlying element references back to somewhere else in the code. For example, within the expression `Account.number` of code section 4.3, variable `number` is considered a reference expression because it is a class field that can reference back to its class, `Account`:

```

1 public class Account {
2     public static int number;
3 }
4
5 public class Transaction {
6     public void transfer() {
7         if (Account.number > 0) {...}
8     }
9 }

```

Code 4.3: Class variable as part of a connasence of meaning

Likewise, referenced expressions can also be local variables or parameters, since they can be traced back to their declaration.

In the case when the underlying element of referenced expression is a **class variable**, we compute the connasence locality by finding the place of declaration of that variable within the system, that is the class where this variable is declared, as well as the enclosing package of its class, and compare these against the class and its enclosing package where the connasence occurs. So for the above example we would compare **Account** and its enclosing package with the class and the package where the **transfer** method is part of, namely **Transaction** and its enclosing package. We set locality according to the distance between these, such that if the classes are the same, a locality of within class is generated, if the classes are from the same package a locality of within package is returned, or otherwise a locality of outside package is assigned to the connasence.

As previously mentioned, another possible example of a reference expression is a **local variable**.

```

1 public void transfer() {
2     Bank myBank = new Bank(SantanderBank.sortCode);
3     ...
4     if (myBank == null) {...}
5 }
6 }

```

Code 4.4: Local variable as part of a connasence of meaning

For example, line number 4 of code section 4.4 has connasence of meaning, since the value of **null** has an implicit meaning which is unclear from the context. For example, it may refer to the fact that as part of additional computation, **myBank** has suffered a transaction error and thus it has been invalidated, or it may also mean that a instance of **myBank** could have not been created since the provided sortcode is not recognised.

When the underlying element is a local variable, detection of connasence locality can be very difficult, since it may be masked by methods that initialise the local variable. In our example above, it is easy to reason that the origin of the connasence is located within the **Bank** class, however, when we are dealing with instances like those in code section 4.5, it becomes harder to trace the location of the source component of the connasence.

```

1 public void transfer() {
2     Bank myBank = BankGenerators.generateBankFromSortcode(SantaderBank.sortCode);
3     ...
4 }

```

Code 4.5: Local variable whose origin is unclear

When dealing with instances like those in code section 4.4, our plugin attempts to generate the locality by applying a similar algorithm to when the reference expression has the underlying element a class variable. However, if no assignment statement can be easily traced, it defaults to a connascence locality within class, as it assumes the origin to be within the same class. Unfortunately, this does not always provide the correct behaviour, and as shown in the evaluation section, it can fail to get the right answer back. Future work includes our ability to improving our tracing algorithms.

Finally, the last type of an underlying element that can be part of a referenced expression is a **method parameter**.

```
1 public void transfer(Account source) {
2     if (source.accountNumber > 10) {...}
3 }
```

Code 4.6: Method parameter which is part of a connascence of meaning

To find the locality of such a connascence, we run a series of different checks on the encapsulation level of the method where the parameter is declared in, as well as checks on the encapsulation level of the classes where the method is defined. The rationale behind checking these is that the encapsulation level is the driver for where the parameter may come from within the system. For example, if the method is public within a public class, it means the parameter may come from a totally different package, as the class and method may be used there, but if the class is for example private, the parameter can only come within the scope of that class.

The algorithm aims to identify the most restrictive encapsulation level used by either method, or the class/classes (if declared within inner class(es)) where the method is defined. If, as a result of this check we find that the encapsulation is set to private (as in code 4.7), we set locality to within class because this is the only scope as permitted by compiler where the method can be used. Note that this is similar if the method is public, but within a private class, or a package method within a private class.

```
1 public class Transaction {
2     ...
3     private void transfer(Account source) {
4         if (source.accountNumber > 10) {...}
5     }
6     ...
7 }
```

Code 4.7: Connascence of meaning with locality set to within class

Additionally, if the most restrictive encapsulation is package-public (as in code 4.8), we set the locality to within package. This can be either a public method within a package class, or a package method within a public class. For brevity, we omitted the cases when the method is declared within an inner class.

```

1 class Transaction {
2   ...
3   public void transfer(Account source) {
4     if ( source.accountNumber > 10 ) {...}
5   }
6   ...
7 }

```

Code 4.8: Connascence of meaning with locality set to within package

Finally, if the found encapsulation is public (as in code 4.9), we set locality to outside package.

```

1 public class Transaction {
2   ...
3   public void transfer(Account source) {
4     if ( source.accountNumber > 10 ) {...}
5   }
6   ...
7 }

```

Code 4.9: Connascence of meaning with locality set to outside package

Although it is true that if a method can be accessed within package or outside package it does not imply that within the codebase it is actually used in such a wide scope, our analysis uses these heuristics because of our real-time analysis requirement. Had we wanted to find out whether the method was actually used somewhere within the package or outside the package of definition, we could have not provided a real-time analysis.

#### 4.1.1.2 Binary Expression

A possibility for the non-literal operand of a binary expression is, of course, another binary expression.

```

1 public void transfer() {
2   if (currentTime - 3600 > 13 * 3600) {...}
3 }

```

Code 4.10: Binary expression part of another binary expression

Line number 2 of code section 4.10 has two connascences of meaning. One of them is between `currentTime` and `3600`. The reasoning is that this literal can have different meanings, like time in seconds, time in milliseconds etc. Thus, it carries an implicit convention with it.

The other connascence is between the binary expression `currentTime - 3600` and the other side of the equation. The rationale is exactly the same, as `13 * 3600` may stand for time in nanoseconds, time in minutes amongst other possibilities.

Whenever faced with a case when the non-literal expression is another binary expression, we set locality for this connascence again, based on the distance between the source of the non-literal expression, its class of origin and enclosing package, and the class and its enclosing package where the problem occurs.

To find the source of the non-literal expression, we look at whether the binary expression is made up of reference expressions or method call expressions, and apply the same algorithms as presented in their respective sections.

### 4.1.1.3 Method Call Expressions

Perhaps the simplest case within a binary expression is that when the non-literal operand takes the form of a method call expression. For example, `userPermissions` is a method call that is part of a wider binary expression, with the literal operand `0`:

```
1 if (userPermissions(user) == 0) { ... }
```

Code 4.11: Method call part of a connasence of meaning

This is yet another example of a connasences of meaning, as there is an implicit meaning/convention to what `0` actually means. For instance, it may stand for admin privileges, but it may also represent guest access level.

In such cases, to identify the connasence locality we first identify the class where such connasence occurs, along with its enclosing package. Secondly, we find the class where the method used within the method call expression is defined in, along with its enclosing package. This gives us enough information to compute the distance between these components, and thus provide a classifier for locality.

### 4.1.2 Switch Statements

As switch statements are, in essence, syntactic sugar for chained if-statements, we have seen many cases in practice where connasences of meaning are hidden within case statements. For example, in code section 4.12, both line 5 and 6 are associated with connasence of meaning as there is an implicit convention behind the value of literals 1 and 2.

```
1 public class Transaction [  
2   ...  
3   public void transfer(Account account) {  
4     switch(account.type) {  
5       case 1: // business customers  
6       case 2: // regular customers  
7     }  
8   }  
9   ...  
10 }
```

Code 4.12: Connasence of meaning within case statements of a switch expression

Essentially, since a switch expression can be either a reference expression - that is a class variable or a local variable or a method parameter - or a method call, and as these have already been covered within the binary expression section, we refer the reader to the implementation details covered by section 4.1.1.

### 4.1.3 Return Statements

The origin location of most connasences that are the result of a comparison against literals or `null` is within methods that return those literals in the first place. Consider the next example:



```

1 public class Database {
2     private Database() {}
3
4     public static Database getInstance() {
5         if (...) { // connection to database failed
6             return null;
7         }
8
9         if (...) { // disk space is full
10            return null;
11        }
12
13        if (...) { // no associated database exists
14            return null;
15        }
16    }
17 }

```

Code 4.13: Singleton Database class

It is very likely that somewhere else in the system, code that tries to obtain a `Database` instance will check for `null`:

```

1 public class Server {
2     ...
3     public boolean storeClientId(Client client) {
4         Database database = Database.getInstance();
5         if (database == null) // something wrong happened;
6             return false;
7
8         return database.store(client.id);
9     }
10    ...
11 }

```

Code 4.14: Server that uses the database to store client id

Indeed, as we notice from code section 4.14, any instance that wants to ensure that the call to the database instance does not result in a `NullPointerException` must check against `null`. However, the meaning of `null` is ambiguous, as lines 6, 10, and 14 of code section 4.13 all provide different reasons for failure. It is why we say that the comparison between `database` and `null` at line number 5 from code section 4.14 is part of a connascence of meaning.

However, this is something we have already seen before within the binary expressions section, but there we focused on the end connascence, namely on the comparison between the non-literal operand and literal. In this section we focus on the source of such connascences, namely on those methods that return literals in the first place.

It is clear that a `null` return statement is going to be checked against somewhere else in the system, as we have already seen. Likewise, if a method returns literals, these will also be checked against by the same rationale as before. Thus, we detect methods that return literals or `null` and label them as creating connascence of meaning.

The locality we set for such instance of connascence depends both on the level of encapsulation of the method and on the level of the encapsulation of the class(es) where the method is defined - plural for cases when the class where the method is defined is an inner class.

The algorithm follows the same steps as presented in the binary expressions section, when the non-literal operand is a reference expression and the reference expression underlying element is a parameter. In rough terms, we find the most restrictive encapsulation level, which may come either from the method or the class(es) where this is defined and set locality with respect to it. For example, if we talk about a public method in a class with package visibility, the most restrictive encapsulation level is of that of class, as although the method is public and could potentially be used anywhere in the system, the class is package visible only, meaning that only components within the same package can access it. In addition, if the most restrictive level is private, we set locality to within class, and for public access we return an outside package locality.

```

protected String entity (String name) {
    if (name.equals("<")) return "<";
    if (name.equals(">")) return ">";
    if (name.equals("&")) return "&";
    if (name.equals("&#35;")) return "#";
    if (name.equals("&quot;")) return "\"";
    if (name.startsWith("&#x")) return Character.toString((char)Integer.parseInt(name.substring(2), radix: 16));
    return null;
}

```

[Connascence of Meaning] Method returns null, which has an implicit meaning more... (§F1)

Figure 4.1: Method `entity` returns `null`, thus it is labelled as connascence of meaning

The figure 4.1 is a screenshot of our plugin analysis of method `entity`, from the `libgdx` [42] framework that we have used for our evaluation. In this example we can observe how the `null` value is highlighted using a yellow colour, since the locality of this type of connascence is set to within package.

#### 4.1.4 Methods Calls

The last identification pattern for connascences of meaning employed by our detection algorithms is checking for literals passed to methods calls, as these very often carry with them an implicit meaning. For instance, some methods accept boolean flags which trigger a different behaviour based on the truth value passed, while in other cases literals such as raw strings can be sent instead as flags.

```

1 public int sort(List<Integer> number, boolean ascending) {
2     if (ascending) //sort ascending
3     else //sort descending
4 }

```

Code 4.15: Sorting is performed with respect to the boolean argument passed

Based on the signature of `sort`, we expect to see usages of it that invoke either an ascending or descending sorting, depending on the value of the boolean parameter. However, since the meaning of passing such parameters is unclear from the context, one needs to dig deep within the method implementation (which may or may not be available, depending whether the method is part of a library or not) to find out which algorithm is triggered. Therefore, we classify such parameters as connascences of meaning. Put simply, there is a convention in the method implementation that programmers must know when using them.

```
1 public Date fromTime(int time) {...}
2
3 public void computeDaysToDate() {
4     Date endDate = fromTime(4200);
5     ...
6 }
```

Code 4.16: Raw literal passed as a parameter to `fromTime` method

Within the code section 4.16, the meaning of the 4200 literal passed to the `fromTime` method is unclear. It may represent days, minutes within other possibilities. Thus, since this carries an implicit meaning, it is also flagged as a connascence of meaning.

Setting locality for such connascences is done following a very similar algorithm to that presented within the return statements section, where we determine the most restrictive encapsulation level of the method and the class(es) in which it is defined - plural for the cases where the method is defined within an inner class - and set locality accordingly.

## 4.2 Refactoring

There is no such code without connascence [4], because a connascence essentially stands for a coupling relationship, and somewhere in the code we must reuse other defined methods. However, not all connascences are created equal. We have already talked about the strength property of connascences 2.4.2, and how there is a scale from the weakest type of connascence, namely of name, to the strongest, namely of identity. One way to proceed with refactoring code with connascence is to try to replace the connascence with one lower on the strength scale. For example, as we are targeting connascences of meaning, and as this is the third element on the scale of connascences, there are only two other possibilities for us to choose from, namely connascence of type and connascence of name.

To extend the utility of our tool, we have also implemented automatic refactorings for some of the problems, that may help users quickly fix detected connascences from their code. Of course, as connascences of meaning are all about implicit conventions, we cannot cover all possible cases, nor sustain that we are able to magically understand the intent behind user code all the time. However, our refactorings are based on heuristics that are often true when similar code errors are found in practice. Put simply, our provided refactorings are appropriate most of the time, but not always.

Presented refactorings are grouped together within a few categories, since although there are various patterns where connascences of meaning can be identified, amongst those there are similar fixes that can be applied. Thus, we present the generalised solutions that are available to users, without reiterating all sections presented in the identification part.

### 4.2.1 Extract literal

Possibly one of the most common refactoring is that of extracting the literal as either a class or local scope constant, for example within the enclosing method. The rationale behind suggesting this refactoring is that by moving to a named constant, we decrease the strength of our connascence, namely we go down on the scale of connascence from meaning to name, which is the weakest type of connascence.

In figure 4.2 we can observe how the value of `index` is compared to `-1`. However, once again, this value carries with it an implicit meaning, as it can stand for a lot of different things, like `index`

```

public int put (K key, V value) {
    int index = indexOfKey(key);
    if (index == -1) {
        if (size < keys.length)
            index = size;
    }
    keys[index] = key;
    values[index] = value;
    return index;
}

```

Figure 4.2: Suggested refactorings for one type of connascence of meaning

```

public int put (K key, V value) {
    int index = indexOfKey(key);
    if (index == INDEX_OUT_OF_BOUNDS) {
        if (size == keys.length) resize();
        index = size++;
    }
    keys[index] = key;
    values[index] = value;
    return index;
}

```

Figure 4.3: Refactored code with literal now a class constant

out of bounds or key not found. Therefore, the suggested refactorings are to extract this literal, `-1`, as either a class or local constant.

In figure 4.3 the first suggestion has been applied, and now `-1` is set to be a class constant, namely `INDEX_OUT_OF_BOUNDS`. Therefore, as we can observe from the figure, there is no highlighting as we have now essentially removed the connascence of meaning, and introduced a connascence of name instead, which is, as explained previously, one of the ways to proceed at targeting connascence refactorings, since the latter connascence is weaker than the former.

#### 4.2.2 Wrap expression inside Optional

Another possible refactoring is one which targets comparisons against `null`. In such cases, we can eradicate the problem if the real intent behind code is to express the absence of a value, which is normally what programmers use `null` for. Of course, there are other cases like we have seen before in code section 4.1.3, where `null` can stand for a variety of different other things, such as database connection error.

In figure 4.4, method parameter `key` is compared against `null`, therefore it is classified as connascence of meaning. Recall from section 4.1.1 that if the non-literal of the binary expression comparison to a literal is a method parameter, then locality for this connascence is set according to the most restrictive encapsulation level between the enclosing method and the class(es) where the method is defined. In our case, `get` is a public method within the public `ArrayMap` class. Therefore, locality is set to the highest possible value, outside package, as this is a candidate for a method that may be used anywhere in the application, hence the red highlighting.

In figure 4.5, the `null` comparison is gone, instead we wrap the non-literal operand within an `Optional` type, and then check whether there is any value associated with that `Optional`. Of course, this only makes sense in cases when we are checking whether a value is present, and may not make sense, as mentioned before, in cases when we are checking whether say a database error occurred. However, as the majority of such comparisons are made specifically targeting the existence of a value, this refactoring is still very useful. Later on in the evaluation section we specifically present data that verifies our assumption about the utility of this refactoring.

#### 4.2.3 Wrap method return type with Optional

Within the last presented refactoring, we have actually included two refactorings as they usually work as a chain, that is most users apply them one after another.

The first refactoring is helping users refactor methods that return `null`, by wrapping the return type with an `Optional` type. As mentioned in section 4.1.3, if a method returns `null`, it is

```

public V get (K key) {
    Object[] keys = this.keys;
    int i = size - 1;
    if (key == null) {
        for (
    } else {
        for (
    }
    return nu...
}

```

Figure 4.4: Suggested refactoring for connascences of meaning that are result of a comparison against `null`

```

public V get (K key) {
    Object[] keys = this.keys;
    int i = size - 1;
    if (Optional.ofNullable(key).isPresent()) {
        for (; i >= 0; i--)
            if (keys[i] == key) return values[i];
    } else {
        for (; i >= 0; i--)
            if (key.equals(keys[i])) return values[i];
    }
    return null;
}

```

Figure 4.5: Refactored code that checks whether the value of `key` exists

extremely likely that any users of it who want to avoid `NullPointerException`s will check against `null`. As more often than not the convention used for `null` is to represent the absence of value, our refactoring targets specifically those cases, by forcing the method to return an `Optional` type, thus giving the possibility of a method to return nothing, that is an `Optional.empty()`, instead of `null`. This helps because whenever users of a method that returns an `Optional`-wrapped type want to check against the absence of value, they can simply do so by invoking calls to the `isPresent` method:

```

1 Optional<Tree> parent = node.getParent();
2 if (!parent.isPresent()) {
3     //invoke error handling procedures
4 }

```

Code 4.17: Intention is clearly defined if methods like `isPresent` are used instead of `null` comparisons

In code section 4.17 the intention behind the check within the conditional statement is unambiguous, as opposed to if the statement within the conditional had been a comparison against `null`.

Of course, we have already seen instances of methods that use different conventions for the meaning of `null`, like connection errors etc. In such cases, perhaps other refactorings like using the `Null Object` pattern may be more appropriate [37]. However, our plugin is currently unaware of other possibilities, but those other possible refactorings are part of our future work, as mentioned within the conclusion section 7.4.

To illustrate this refactoring with real instances of code, as well as to link our description to the next part of this refactoring chain, we present the following figures:

```
public VertexAttribute findByUsage (int usage) {
    int len = size();
    for (int i = 0; i < len; i++)
        if (get(i).usage == usage) return get(i);
    return null;
}
```

! Wrap the return type of the method with Optional<T> ▶

Figure 4.6: Method which returns `null`, detected as connascence of meaning of highest severity

Part of figure 4.6 is our plugin’s analysis over the `findByUsage` method. Since the method returns `null`, and the encapsulation level of method is public, as well as its enclosing class, `VertexAttributes`, the plugin correctly detects this as an instance of connascence of meaning and associates the locality for it as outside package, making this method an ideal candidate for refactoring as it is part of the most severe errors. The suggested refactoring for this connascence of meaning is, as previously mentioned, to wrap the return type of the method with the `Optional` type.

```
public Optional<VertexAttribute> findByUsage (int usage) {
    int len = size();
    for (int i = 0; i < len; i++)
        if (get(i).usage == usage) return get(i);
    return null;
}
```

! Replace null with Optional.empty() ▶

Figure 4.7: Intermediate refactoring step that wraps method return type with `Optional` type

Once the previous refactoring is applied, we can notice from figure 4.7 the return type has changed from `VertexAttribute` to `Optional<VertexAttribute>`, but the `null` return statement has remained unchanged. However, the new presented refactoring essentially targets this problem by suggesting to replace the old `null` return with an `Optional.empty()`, that is to make the implicit convention used for `null` explicit, which in our case is the absence of a value. Additionally, we can also observe that since the method return type has changed, the other return statement is now part of a compilation problem. Indeed, the return type of `get(i)` method is of type `VertexAttribute` and not `Optional<VertexAttribute>`.

```
public Optional<VertexAttribute> findByUsage (int usage) {
    int len = size();
    for (int i = 0; i < len; i++)
        if (get(i).usage == usage) return get(i);
    return Optional.empty();
}
```

Figure 4.8: After “replace null with `Optional.empty()`” refactoring is applied

The last two figures, 4.8 and 4.9, show the outcome of applying the second suggested refactoring, thus replacing `null` with `Optional.empty()`, as well as correcting the previous-introduced compile error, thus wrapping the invocation of `get(i)` within an `Optional.of()` method to make its return type conform to the new return type of `findByUsage` method.

```
public Optional<VertexAttribute> findByUsage (int usage) {  
    int len = size();  
    for (int i = 0; i < len; i++)  
        if (get(i).usage == usage) return Optional.of(get(i));  
    return Optional.empty();  
}
```

Figure 4.9: Final version of method `findByUsage` without connascence of meaning or compilation errors

Once again, the outcome of this chain of refactoring is that we have removed a potentially severe connascence of meaning as presented in figure 4.6, and enabled users of this method with the possibility of explicitly, as opposed to implicitly, checking whether the object returned exists or not.

## Chapter 5

# Connascence of Position

Software components have connascence of position if they must agree on the order of values. Consider the following example:

```
1 public int [] getUserDetails() {
2     int [] details = new int [2];
3     details [0] = getUserId ();
4     details [1] = getUserAccessLevel ();
5     return details;
6 }
```

Code 5.1: Method which retrieves id and access level of a particular user

Based on the return type of the `getUserDetails`, somewhere else in the system there will be usages of it that specifically target either the id or the access level by indexing within the array:

```
1 public void executeScript () {
2     int [] user = getUserDetails ();
3     if (userHasExecutePermissions (user [1])) // execute script
4     else // register attempt from user id, user [0]
5 }
```

Code 5.2: Method that targets specific values within the retrieved array from `getUserDetails`

Within the `executeScript` method of code 5.2, lines 3 and 4 contain connascences of position because the code assumes a specific ordering of values within the `user` array, namely that the permission access level is the second element, while the user identity is the first.

Arguably, such instances of code are not common in object-oriented code anymore, as they reassemble more the way people wrote code in a structured language such as C. However, that code still compiles and may exist even in Java, C++ and other languages.

Connascences of position are not limited to old(er) programming languages, and the following example is one of the most common ways in which such problems can arise within object-oriented code:



```

1 public class EmailServer {
2     ...
3     public static void sendEmail(String firstName, String lastName, String subject,
4         String body, String address) {
5         ...
6     }
}

```

Code 5.3: Method with multiple arguments of the same type

With such a method, it is easy to realise that if developers are not careful about the ordering of the parameters they pass, they can get the signature wrong. For example, it is very easy to mistake the location of any argument, as any combination would make this code compile since all are of the same type. Presumably swapping the order of `firstName` and `lastName` would not have a very big impact, but depending on importance of the email, mistaking `address` for `body` may well have a big impact. The whole problem with such code is that the compiler cannot enforce users to pass the right argument at the right position, and since the ordering of these values is important we classify this method as having connascence of position.

Although not quite found within so many different patterns as the connascence of meaning, connascence of position is a stronger connascence, as measured on the connascence strength scale, especially since usually the consequence of it, as seen in code section 5.3, may be more dramatic than those resulted from using connascence of meaning. After all, a `NullPointerException` is normally a less serious offence than potentially sending a confidential email to the wrong person.

In the rest of this chapter we discuss the identification patterns where such connascences arise from, talk about measuring the locality property in each instance, and conclude by presenting our proposed refactorings that help users mitigate such problems.

## 5.1 Identification

### 5.1.1 Methods and Constructors

One of the most often seen examples of connascences of position in code is, as seen in code section 5.3, when a method parameter list contains multiple elements of the same type. The order in which those arguments appear need not be adjacent:

```

1 public Window generateWindow(int height, Colour frameColour, int width) {...}

```

Code 5.4: Connascence of position from method that takes multiple arguments of the same type

Users of `generateWindow` method depend on the ordering of `height` and `width`, as passing the wrong way of arguments around, that is `width` instead of `height` and vice-versa, would not result in the desired outcome. However, even though they are not adjacent, this is still identified as a connascence of position.

Moreover, when looking for methods that take multiple arguments of the same type, we make no distinction if the repeating type is a primitive or not. For example, code section 5.3 presents a method linked with connascence of position where the repeating type is a non-primitive, and code section 5.4 presents a method linked to connascence of position where argument type is, in fact, a primitive.

Constructors are yet another place where users may pass multiple arguments of the same type, and code such as that from section 5.5 is more often than not part of our software.

```

1 public class User() {
2     ...
3     public User(String firstName, String lastName) {...}
4     ...
5 }

```

Code 5.5: Constructor that takes multiple arguments of the same type

To compute the locality of such connascences of position, we identify the most restrictive encapsulation level between the method/constructor privacy level and its enclosing class or classes, if defined as part of an inner class. For instance, if we are dealing with a public method within a class that has visibility set to package, the most restrictive encapsulation level is that of class, namely package. Similarly, a public method within a private class results in a private most restrictive encapsulation level.

We therefore set the locality property based on the most restrictive encapsulation level, such that a locality of within class is assigned to private encapsulation, as it means the method/constructor may be only used within that class. For package-level encapsulation we set a locality of within package, and for public encapsulation we set it to outside package, as it may be used anywhere in the system.

Due to our requirements of real-time analysis, we are unable to verify whether a method with a package or public encapsulation are actually used within the package or the rest of the application, as this type of detection would take more time than available for our analysis to perform in real-time. This is however one area that we plan to add future improvements, and is part of our planned future work as mentioned in the conclusion section.

```

1 public class Server {
2     ...
3     public Server(int socket, int inet4) {...}
4     ...
5 }

```

Code 5.6: Connascence of meaning with locality set to outside package

### 5.1.2 Methods Calls

Another pattern for identification of connascences of position is directly within code that makes use of similar type of methods with those presented in the previous section. For example:

```

1 public class EmailClient {
2     ...
3     public sendEmail() {
4         String firstName = "John";
5         String lastName = "Smith";
6         String address = "john.smith@ic.ac.uk";
7         String subject = "Final_Report";
8         String body = "Our_report_has_been_uploaded_to_CATe";
9         EmailServer.sendEmail(firstName, lastName, address, subject, body);
10    }
11    ...
12 }

```

Code 5.7: Wrong usage of `sendEmail` method

In the above example, we assume that both `EmailClient` and `EmailServer` of code section 5.3 are part of the same package. Therefore, we identify the method call `sendEmail` from line 9 as having connascence of position. The reason is that users of such method depend on the ordering of values passed to the method. It is clear how such orderings are dangerous, and our example emphasizes this by passing the `address` as the third parameter, as opposed to the last, as it should have been. After all, it only takes a moment of distraction for such an error to occur.

Our identification algorithm works by detecting the underlying signature of the method call and, much like discussed in the previous section, analyse whether it takes duplicate types.

However, it need not be the case that the actual arguments have the same type per se. As an example, consider the following scenario where argument types are mixed between superclasses and subclasses:

```
1 public class Account() {...}
2
3 public class BusinessAccount extends Account() {...}
```

Code 5.8: Chain of inheritance of `Account` class

```
1 public class Transaction {
2     ...
3     public static void transfer(Account to, BusinessAccount from) {...}
4     ...
5 }
```

Code 5.9: Bank transaction utility class

Based on the definition of the `transfer` method of `Transaction` class, as well as taking into account the inheritance tree of `Account`, users are faced with the possibility of passing the arguments in the wrong order. For example:

```
1 public class Payments {
2     ...
3     public void paySalary(BusinessAccount source, BusinessAccount destination) {
4         Transaction.transfer(source, destination);
5     }
6
7     public void paySalary(BusinessAccount source, Account destination) {
8         Transaction.transfer(source, destination);
9     }
10    ...
11 }
```

Code 5.10: Incorrect usages of `transfer` method

The methods within the `Payments` class are a prime example why it is not always sufficient to only check that methods take multiple arguments of the same type. In the first instance of the `paySalary` method, it is clear that the order of arguments passed to the `transfer` method is actually flipped. Based on the definition from code section 5.9, our `paySalary` method is going to take money from the `destination` account, and transfer them into the `source` account, whereas the expected behaviour in this case is the complete opposite. As this code compiles, there is usually no additional help that users can benefit from to help them identify such subtle bugs, unless thorough testing is performed. However, our inspection detects that the `transfer` method actually receives

two arguments of the same type and therefore, it detects this is as an instance of connascence of position.

For simplicity, we assume that the `Payments` class is located within the same package to the `Transaction` class, so the locality of such connascence is reported as being within package and thus, according to our highlighting scheme, the name of the offending method is surrounded with yellow colour.

The overloaded version of `paySalary` method is a different example to the first one, because unlike the first version, this code does not compile, hence why it is underlined to represent a compilation problem. The reason why such code does not compile is because the order of arguments is wrong, and as `destination` has `Account` type, it cannot match the expected `BusinessAccount` expected by the `transfer` method. This is an instance where the compiler can actually notify users about potential problems with the ordering of arguments, however, as seen in the previous example, it is not always that a compiler can catch such usages.

Furthermore, the type of arguments where such connascences of position can occur include primitives as well. For instance, if a method takes arguments of distinct primitive types, say an integer and a float, a similar situation like that in the first version of `paySalary` can arise.

```
1 public class WeatherDatabase() {
2     ...
3     public void storeWeather(int time, float temperature) {...}
4     ...
5 }
```

Code 5.11: Class stores details about weather temperatures as measured at a specific UNIX time

Based on the definition of `storeWeather` method, erroneous usages can easily appear within the system.

```
1 public class WeatherMonitoringSystem() {
2     ...
3     public void addWeatherRecord(int time, int temperature) {
4         WeatherDatabase database = fetchWeatherDatabase();
5         database.storeWeather(temperature, time);
6     }
7     ...
8 }
```

Code 5.12: Wrong usage of `storeWeather` method

As per the example of `paySalary`, within the `addWeatherRecord` the order of arguments passed to the `storeWeather` is mistaken. In addition, as the compiler does not detect anything wrong with such code, only a very thorough testing suite would catch such mistake. The result of such an error can have a lasting impact, as for instance, weather forecasts based on the the stored information may retrieve temperature values completely out of scale. However, *Connascer* is able to detect this usage of `storeWeather` and alarm the user about potential errors. The connascence locality set for this connascence is once again, within package, as we assume that both `WeatherMonitoringSystem` as well as `WeatherDatabase` are within the same package.

In general, to compute the locality property of connascences of position which are a result of method calls, we first identify the location where the connascence takes place in terms of class and its enclosing package, and then look up the class where the method is defined in, as well as its enclosing package. Finally, we compare these elements to identify how close or far away in the

system - within the same class, within the same package, outside the package - these elements are and then assign the locality accordingly.

### 5.1.3 “new” Expressions

As mentioned within section 5.1.1, constructors can generate connascences of position if they take multiple arguments of the same type. As invocations of methods that take multiple arguments can bring new connascences of position to the system, so too can instantiation of objects that are based on constructors which take multiple arguments of the same type.

```
1 public class Window {
2     ...
3     public Window(int width, int height) {...}
4     ...
5 }
6
7 public class UI {
8     ...
9     public void setUpMainUI() {
10         Window mainWindow = new Window(someWidth, someHeight);
11     }
12     ...
13 }
```

Code 5.13: Connascence of position resulted from a “new” expression

In the code section 5.13 we assume that both `Window` and `UI` are within the same package, and omit the highlighting that would normally be associated with the constructor within the `Window` class, as examples of that have already been shown within section 5.1.1. However, within the `UI` class, we have an instantiation of a `Window`, which uses the same constructor that takes multiple parameters of the same type, and therefore we introduce a new connascence of position. Its locality is set to within package based on our earlier assumption.

Since “new” expressions are very similar to method calls discussed in section 5.1.2, we refer from going into further details about our identification process. However, we wanted to let readers know about the possibility of such cases.

## 5.2 Refactoring

To help developers improve the quality of their code, *Connascer* also adds support for automated refactorings that fix connascences of position. Throughout the identification section we presented different patterns where such connascences can arise, and now we present two refactorings that help users remove connascence of position problems from their method/method calls as well their constructor/constructor invocations.

One way to refactor connascences is by introducing weaker connascences, as measured on the strength scale. In the case of connascence of position, weaker connascences include connascence of name, type, meaning, and algorithm, in ascending order of strength. Out of these four possibilities, an elegant solution is to introduce replace the connascence of position with a connascence of type, that is to move from depending on the order of values to depending on the type of values. Such a refactoring is important because if types do not match, it is easy for compilers to flag such errors, thus preventing potential mistakes as shown in the identification section.

Please note that all examples used in this section are generated from analysis over the `libgdx` library.

### 5.2.1 Introduce Parameter Object

When a method or a method call is flagged as connascence of position, the available option within Java to migrate to the connascence of type is to collect all those arguments that are of the same type, and wrap them inside a parameter object [9]. For example:

```
744 Draws a rectangle for the bounds of this actor if {@link #getDebug}
protected void drawDebugBounds (ShapeRenderer shapes) {
    if (!debug) return;
    shapes.set(ShapeType.Line);
    shapes.setColor(stage.getDebugColor());
    shapes.rect(x, y, originX, originY, width, height, scaleX, scaleY);
}
```

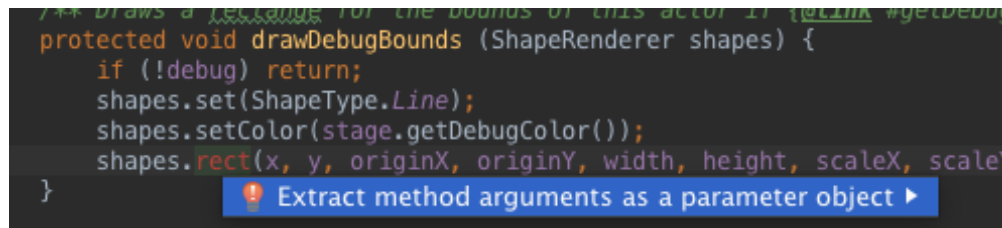


Figure 5.1: Method `rect` takes nine(!) parameters, all of type `float`

Within the `drawDebugBounds` method, the usage of `rect` method on the `shapes` parameter is flagged as a connascence of position, and the red highlighting is an indicator that this its locality is set to outside package. Indeed, analysis within the codebase reveals that both `drawDebugBounds` class, `Actor`, and `ShapeRenderer` are in totally different packages. The suggested refactoring is to extract the method arguments as a parameter object. However, as there are nine distinct parameters, a single object is insufficient as most of these can be grouped in couples, apart from the last.

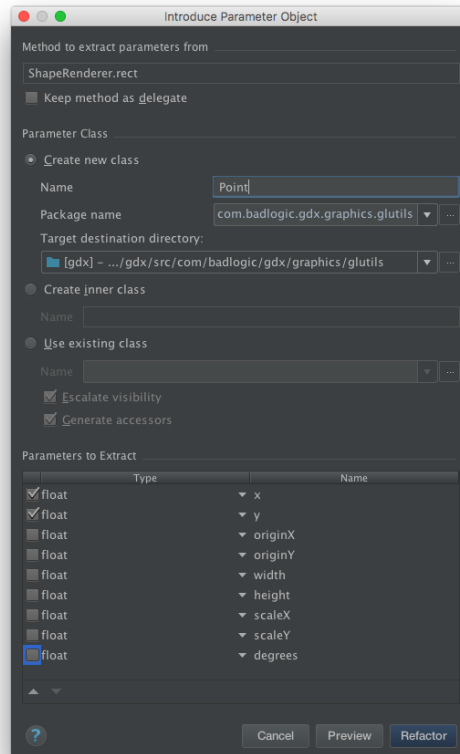


Figure 5.2: Introducing a Point object for parameters x and y of rect method

Thus, as presented within figure 5.2, we apply the refactoring for each pair of parameters which fit together within the same object. This process involves introducing a parameter object, **Point** for x and y arguments, a parameter object **Origin** for **originX** and **originY** argument, a parameter **Dimensions** for arguments **width** and **height**, and finally, a parameter object **Scale** for **scaleX** and **scaleY** arguments. The end result is presented in the next figure.

```
protected void drawDebugBounds (ShapeRenderer shapes) {
    if (!debug) return;
    shapes.set(ShapeType.Line);
    shapes.setColor(stage.getDebugColor());
    shapes.rect(new Point(x, y), new Origin(originX, originY), new Dimensions(width, height), new Scale(scaleX, scaleY), rotation);
}
```

Figure 5.3: Refactored rect method of ShapeRenderer class

One thing to notice with this type of refactoring is that we do not need to be within the same class where a method that takes multiple arguments is defined, in our case we did not need to be within the **ShapeRenderer** method to perform a refactoring on the **rect** method.

Furthermore, it should be also noted that within this refactoring we went from a method that took nine parameters of the same type, where a chance of accidentally passing some in the wrong order was high, to one where it takes five parameters, all of different types. Although even five parameters is arguably a high number of arguments to pass to a method, notice that since all are distinct it is impossible to mistake the order of them.

Finally, the reasoning behind why newly-introduced objects, `Point`, `Origin`, `Dimensions`, and `Scale` are highlighted as connascences of position with a severe locality is because each one of these constructors takes two arguments of the same type, `float`, and when we introduced each class within the system we placed them in a different package to the `Actor` class's enclosing package, which is the class where method `drawDebugBounds` is declared. Arguably we could have avoided this by placing them within the same package, but then each constructor invocation would have still been highlighted as a connascence of position, albeit with a not-so-critical severity. We fix remaining problems within the next refactoring.

### 5.2.2 Replace Constructor with Builder

Although to refactor the case when a constructor takes multiple parameters of the same type we could potentially apply the same “introduce parameter object” for those arguments that have the same type, and technically resolve the connascence, we could not opt for this solution as the new object would have been faced with the same problem, namely that its constructor would be flagged as a connascence of position because it would take multiple arguments of the same type, thus when applied to a constructor this refactoring only keeps us going in loops, without really solving the problem.

One approach that solves the connascence of position problem of constructor is the builder pattern [9]. Since a builder constructs the object by passing each method only one argument, we cannot talk about connascence of position. Therefore, *Connascer* suggests moving the constructor to a builder pattern in such cases.

To build on our previous example, we chose to refactor the constructor of the `Point` object as it was the first in the list of arguments.

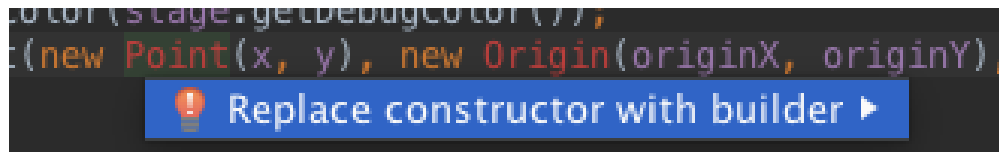


Figure 5.4: Suggested refactoring for constructor with connascence of position

We trigger the suggested refactoring to remove the connascence of position associated with `Point`, and move to a builder pattern.



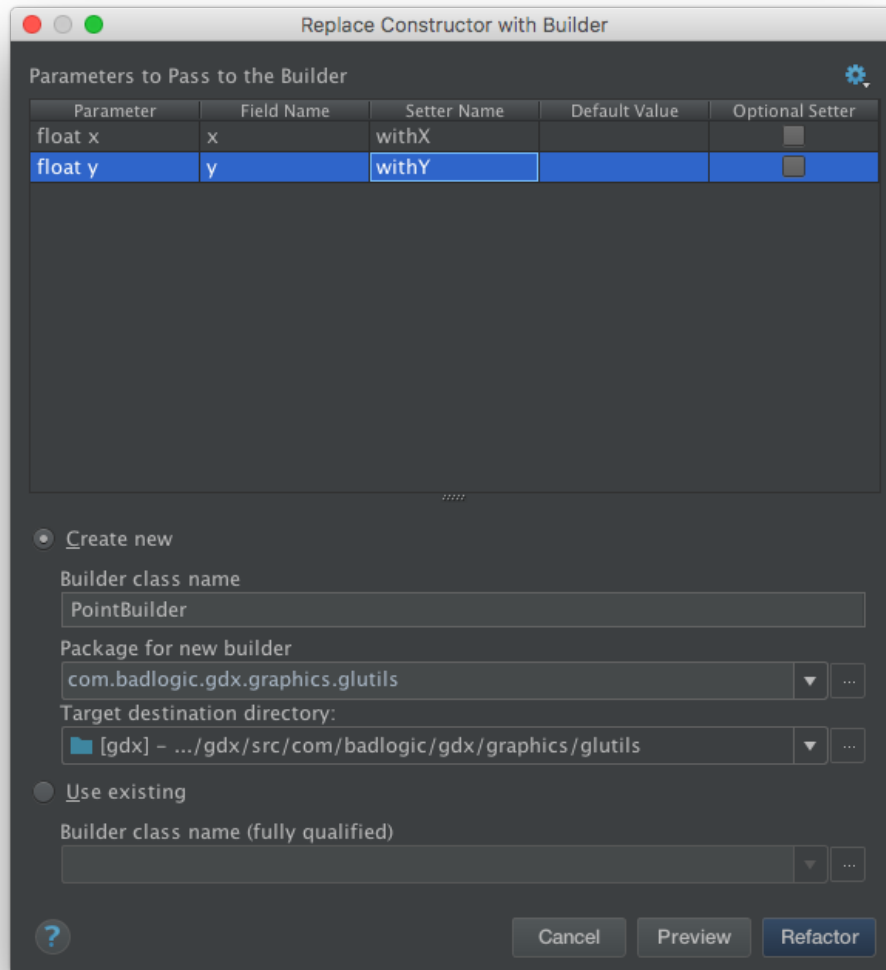


Figure 5.5: Builder interface - we customise the name of setter methods to `withX` and `withY`

Once the `PointBuilder` class is created, all usages of the `Point` constructor are replaced with delegated calls to the newly-constructed builder, thus removing all associated connascences of position.

```
protected void drawDebugBounds (ShapeRenderer shapes) {  
    if (!debug) return;  
    shapes.set(ShapeType.Line);  
    shapes.setColor(stage.getDebugColor());  
    shapes.rect(new PointBuilder().withX(x).withY(y).createPoint(), new Origin(originX, originY),  
}
```

Figure 5.6: Old instantiation associated with connascence of position is replaced by delegate methods to `PointBuilder` builder

To fully refactor the `rect` method, the same process must be applied for the other introduced parameter objects.

In the end, our process refactored a method which took nine parameters of the same type to one that only took almost half, removed all possibilities of mistaking the order of arguments by introducing object parameters of different types, and finally, refactored the newly introduced parameters to builder patterns in order to guarantee that the ordering of arguments when instantiating the newly-introduced parameter objects is irrelevant.

## Chapter 6

# Evaluation

To evaluate *Connascer* we present several sections that aim to take a look through different aspects. Namely, we start off by evaluating how accurate the plugin is by running it on a significant Java codebase and comparing actual results with the expected output. We then take a look at various indicators of its performance. Furthermore, we investigate plugin's functionality by analysing results obtained from users through live demonstration sessions, usage monitoring and user review. We conclude with remarks highlighting strengths and weaknesses and putting them in balance.

Additionally, it should be mentioned that some pieces of evaluation like accuracy were tested on one of the latest stable iterations of *Connascer*, while some others like usability were the result of a continuous evaluation through different iterations of the plugin.

### 6.1 Accuracy

#### 6.1.1 Evaluation method

To learn about the accuracy of our detection plugin, we decided to evaluate it on an appropriate codebase. We chose `libgdx`, a cross-platform gaming engine that over the time has won various awards [42]. The codebase has seen over well over 12500 commits, from over 380 contributors, and more than 42 stable releases over its 8 years of existence, making it thus an ideal candidate to detect potential problems in a software that has seen so many twists and changes over its lifetime. Additionally, since the community behind is still very active, our analysis on the software quality is highly relevant as the platform is expected to be in a continuous evolution.

Ideally, to perform a high accuracy analysis one would need to compare the results of our plugin against the expected results. However, as there are no other tools available that detect connascence, our expected results are generated by manual review of source code. Given this limitation, we were unable to compare all obtained results from the plugin analysis as the required time to compute the expected results would have been in the order of weeks due to the large size of the codebase. Therefore, to evaluate the results of connascence inspection we decided to randomly select 100 reported problems and evaluate those components where the problems originated from against expected results.

#### 6.1.2 Connascence of Meaning

As mentioned in section 4.1, software components have connascence of meaning if they must agree on the convention of particular values, for example the convention used to represent error codes for a utility function, which may be represented in some cases as `-1` or in other cases as `null`.

Additionally, as discussed in section 2.4.3, our analysis puts emphasis on the `locality` of a connascent instance, so that that more important connascentences (those that are further apart in the system) are highlighted with a higher severity. That translates in underlying connascentence instances that are part of the same class, highlighting with yellow those connascentences within the same package but not class, or flagging with an intense red, similar to compilation errors, connascentences that cross package boundary.

### 6.1.2.1 Identification

Our random selection of software components to evaluate the identification process of connascentence of meaning included components under the `com.badlogic.gdx.utils` and `com.badlogic.gdx.scenes` packages from the `libgdx` framework.

From the analysed reported instances of connascentence of meaning, we were unable to rule out any as false positive. Although our goals involved delivering a low false-positive reporting plugin, this result is more likely to be a consequence of the rather small testing population, which may not exhibit the outliers that may determine *Connascenter* to report errors incorrectly. Therefore, further experiments are necessary to have a better understanding of this part of evaluation. Our planned follow up test is to gather data from as many codebases as possible, to enlarge testing population, and thus capture some of the outliers that our initial test data was missing.

However, unlike false positives where the testing data might have not been revealing enough, our evaluation has found a series of false negatives, namely 13 missed connascentences of meaning. We found that those cases could be grouped together into a few clusters, thus we present a generalised format of those missed problems.

From the list of the problems that were not labelled as connascentences of meaning, the most encountered issues was when there was an assignment between a variable and some magic number. For example, consider the following code:

```
1 public static int numChars (int value, int radix) {
2     int result = (value < 0) ? 2 : 1;
3     ...
4     return result;
5 }
```

Code 6.1: Method of `StringBuilder` class from `com.badlogic.gdx.utils` package

In this method, the variable `value` is compared against 0, thus implying an underlying meaning for result of the comparison. Therefore, as mentioned in section 4.1, our algorithm correctly detected this instance as connascentence of meaning. However, *Connascenter* stopped short of detecting the other connascentence, namely the assignment of 1 and 2 to `result`. Such cases can be left undetected primarily because the assignment happens indirectly, as a result of another connascentence of meaning. The correct behaviour would have been to report this connascentence by underlining both literals since they are part of the weakest type of connascentence locality, namely within the same class.

```
1 public static int numChars (int value, int radix) {
2     int result = (value < 0) ? 2 : 1;
3     ...
4     return result;
5 }
```

Code 6.2: Corrected connascentence of meaning detection for `numChars` method

Another problem identified was that of mislabeled connascence locality. For example, consider this code:

```
1 public void remove () {
2     Tree tree = getTree();
3     if (tree != null)
4         tree.remove(this);
5     else if (parent != null)
6         parent.remove(this);
7 }
```

Code 6.3: Method of `Tree` class from `com.badlogic.gdx.scenes.scene2d.ui` package

The plugin flagged lines 3 and 5 as potentially containing connascence of meaning, with the locality set to the weakest instance in both cases. Although this is true for line number 5, as `parent` is an instance variable of `Tree` class, and therefore the comparison against `null` is rightly labelled with a weak locality, things are different for the reported problem at at line number 3.

The `tree` local variable is initialised as the result of `getTree()` method. Further analysis in the codebase revealed that this method generates an instance of type `Tree` that is the result of accessing a member in the `Actor` class within the same package. Therefore, the correct behaviour here would have been to detect the locality of this connascence of meaning as within package, as opposed to within class. Such mislabelings can happen when the source of a local variable is unclear. Here, the `remove` method is not responsible for initialisation of local variable `tree` by, instead it delegates this task to another method.

```
1 public void remove () {
2     Tree tree = getTree();
3     if (tree != null)
4         tree.remove(this);
5     else if (parent != null)
6         parent.remove(this);
7 }
```

Code 6.4: Corrected connascence of meaning detection for `remove` method

### 6.1.2.2 Refactoring

Out of the 100 detected connascences of meaning, we randomly selected 30 of those to apply suggested plugin refactorings. From these, 24 were successful, but 6 instances failed to resolve the original problems. As per previous section, those unsuccessful refactorings can be clustered together, and without losing generality, we present the representative errors of each cluster.

One of the solution for fixing a comparison against `null` is often to make use of Java's `Optional` class, which wraps around any non-primitive type and can substitute comparisons against `null` with method calls such as `Optional.isPresent()`. Likewise, comparisons against `null` can only arise from methods that return `null` in the first place, thus a sensible suggestion is to wrap the actual return type with an `Optional`, to signify that the method may not return an actual result.

Unfortunately, one of downsides of the `Optional` class is that it is only available in the latest iteration of Java's development kit, that is version 8, which means any older version would not support it. Therefore, a JDK check is incorporated as part of the plugin refactoring suggestion mechanism. However, consider the following code:

```

1 public Actor hit (float x, float y, boolean touchable) {
2     if (clip) {
3         if (touchable && getTouchable() == Touchable.disabled) return null;
4         if (x >= getWidth() || y >= getHeight()) return null;
5     }
6     return super.hit(x, y, touchable);
7 }

```

Code 6.5: Method of class `Table` from package `com.badlogic.gdx.scenes.scene2d.ui`

As the `hit` method returns `null`, the plugin correctly identifies and reports both return statements from lines 3 and 4 as connascences of meaning, and for each it suggests to “wrap return type of the method with `Optional` type”. Although our JDK was set to the latest version, this refactoring resulted in an error because somewhere else in the code the “`@since 1.6`” annotation was used, meaning that this class was developed with JDK version 6, and so users of it would not be expected to find features from newer versions. A way to fix this problem would be to offer more possible solutions like the `NullObject Pattern` that do not require a specific Java JDK version, and to further check against specific annotations such as “`@since`” or “`@version`” that can restrict some of the proposed solutions.

Within code section 6.6, another example of connascence of meaning is presented. In such cases, we are expected to first wrap the return type of the method `getStage()` with `Optional` and then invoke the `isPresent()` method. However, in instances when the source of the problem, that is source file of method, does not have write permissions, this refactoring fails.

```

1 public boolean fire (Event event) {
2     if (event.getStage() == null) event.setStage(getStage());
3     ...
4 }

```

Code 6.6: Method of class `Actor` from package `com.badlogic.gdx.scenes.scene2d`

In the case of `fire` method, the connascence of meaning is correctly detected from the `null` comparison of `getStage()` method, thus the suggestion offered requires changing the signature of the method `getStage()` from the `Event` class by wrapping its return type with the `Optional` class. However, as the permissions set for this class are for read-only, triggering this refactoring results in an error about file permissions. This is one aspect that the plugin does not check, but rather silently assumes that all documents have both read and write permissions. Therefore, a fix would imply adding a permission check within the refactoring suggestion algorithm.

### 6.1.3 Connascence of Position

As discussed in section 5.1, software components have connascence of position if they must agree on the order of values. For example, the order of arguments of a method may be considered connascence of position if the arguments are of the same type. Additionally, like with connascence of meaning, the locality property is also taken into account, and the same three different possible situations, within class, within package and outside package are also applicable.

#### 6.1.3.1 Identification

Our random selection of software components to evaluate the identification process of connascence of position included components under `com.badlogic.gdx.graphics`, `com.badlogic.gdx.assets`

and `com.badlogic.gdx.utils` packages.

As per evaluation of connascences of meaning, we were yet again unable to detect false positives. Once again, we think this is partly due to the rather limited size of our testing sample. Despite being unable to find false positives, along with the 100 tested connascences of positions we have also discovered 9 instances of connascences of position that either went undetected, false negatives, or which have been reported as having a different connascence locality than what it really was.

One common case that we have found amongst those unreported connascences were array/collection accesses. As raw access within an array or collection assumes an implicit ordering of values, these cases should have been flagged as connascences of position. To exemplify, please consider the following:

```
1 public boolean add (int key) {
2     ...
3     int [] keyTable = this.keyTable;
4
5     // Check for existing keys.
6     int index1 = key & mask;
7     int key1 = keyTable[index1];
8     if (key1 == key) return false;
9
10    int index2 = hash2(key);
11    int key2 = keyTable[index2];
12    if (key2 == key) return false;
13
14    int index3 = hash3(key);
15    int key3 = keyTable[index3];
16    if (key3 == key) return false;
17    ...
18 }
```

Code 6.7: Method of class `IntSet` from package `com.badlogic.gdx.utils`

Within the `add` method it is clear that all three indices, `index1`, `index2` and `index3` are targeted at specific locations within the `keyTable` array, thus revealing the underlying assumption for specific ordering of values within this array. A corrected output is presented below.

In the new output, `index1` is underlined, meaning that this is a connascence of position which has a locality of type within class. Indeed, if we trace the origin of the local variable we find that it is the result of a logical and (`&&`) between the class variable `mask` and parameter `key`. However, as the method `add` is only used as a helper method within the same class, and method parameter - `key` - is only used within the same class too, the origin `index1` is set within the same class as a result.

Furthermore, `index2` is now also recognised as a connascence of position, with its locality set to within class. This is because although `index2` is the result of an assignment from the `hash2` method, that method is actually defined locally, within the same class. For brevity we omitted the chunk of code associated with `index3` as it is identical to the one where `index2` is defined.

```

1 public boolean add (int key) {
2     ...
3     int [] keyTable = this.keyTable;
4
5     // Check for existing keys.
6     int index1 = key & mask;
7     int key1 = keyTable[index1];
8     if (key1 == key) return false;
9
10    int index2 = hash2(key);
11    int key2 = keyTable[index2];
12    if (key2 == key) return false;
13    ...
14 }

```

Code 6.8: Corrected connascence of position for add method

Another common problem we found was a mislabelling of various connascences of position locality. Most often the reason behind such incidents was not checking the project scope for usages of those methods identified as having connascences of position. One such case is the following:

```

1 @Override
2 public void resize (int width, int height) {
3     ...
4 }

```

Code 6.9: Method of class `ScreenAdapter` from package `com.badlogic.gdx.graphics`

Method `resize` was correctly identified as having connascence of position, since it carries two arguments of the same type, which users would have needed to know the ordering to avoid calling it erroneously, for example passing `height` instead of `width` and vice-versa. What the inspection did not identify correctly is the locality of this connascence. Further analysis within the codebase revealed this method was used in at least two different locations, however, both were within the same package. Therefore, a corrected reporting for this instance of connascence would have flagged it with a within package locality, as it is used by other components within the module boundary.

```

1 @Override
2 public void resize (int width, int height) {
3     ...
4 }

```

Code 6.10: Corrected connascence of position for `resize` method

### 6.1.3.2 Refactoring

To evaluate the refactoring capabilities of our plugin with respect to connascence of position suggestions, we randomly selected 30 out of the 100 detected connascences of position and tested the impact of refactorings on each of those issues.

Out of 30 refactorings, 25 were successful, and 5 of them failed, placing it almost on the same level of successfulness with refactorings suggested by connascences of meaning analysis. Out of the five distinct failures, we could identify only two of them as having different reasons for failure, thus we shall present these distinct instances.



One solution to fix the problem of connascence of position for a constructor is to replace it with a builder pattern, and our plugin has automatic refactorings to help streamline this process. However, when the constructor is within an abstract class, our refactoring fails because the generated builder tries to instantiate the abstract class. Consider the following:

```
1 public abstract class GLFramebuffer<T extends GLTexture> implements Disposable {
2     ...
3     public GLFramebuffer (Format format, int width, int height, boolean hasDepth) {
4         ...
5     }
6     ...
7 }
```

Code 6.11: Method of class `GLFramebuffer` from package `com.badlogic.gdx.graphics.glutils`

As the constructor of abstract class `GLFramebuffer` has two parameters of same type, namely `width` and `height`, our plugin correctly identified this as an instance of connascence of position, thus highlighting it to represent its connascence of locality of outside package.

However, after applying the suggested refactoring, namely to “replace constructor with builder”, the following is created:

```
1 public class GLFramebufferBuilder<T extends GLTexture> {
2     ...
3     // Omitted setters for various fields
4     ...
5     public GLFramebuffer build() {
6         return new GLFramebuffer(format, width, height, hasDepth);
7     }
8 }
```

Code 6.12: Suggested builder class

Unfortunately, line number 6 does not compile as `GLFramebuffer` is an abstract class, and thus it cannot be instantiated. A solution in such cases would involve a similar approach to how connascences of position are handled when they are associated with methods, but not constructors, namely to extract the arguments as a parameter.

One of the other unsuccessful refactorings relates to the same class of suggestions, namely replacing the constructor of a non-abstract class with a builder. Here, the problem is that although the refactoring goes through, and builder class gets created successfully, the initial constructor, but not its usages, is left unchanged, thus allowing the possibility of new connascences. See following code snippet:

```
1 public class Pixmap implements Disposable {
2     ...
3     public Pixmap (int width, int height, Format format) {
4         ...
5     }
6     ...
7 }
```

Code 6.13: Constructor of class `Pixmap` from package `com.badlogic.gdx.graphics`

Once again, the plugin correctly detects this constructor as having connascence of position, since both `width` and `height` share the same type. Applying its suggestion results in another class, `PixmapBuilder` being created:

```
1 public class PixmapBuilder {
2     ...
3     // Omitted setters for various fields
4     ...
5     public Pixmap build() {
6         return new Pixmap(width, height, format);
7     }
8 }
```

Code 6.14: Builder class for `Pixmap`

Besides creating a new builder class, all instantiations of `Pixmap` are also replaced with:

```
1 PixmapBuilder.withHeight(...).withWidth(...).withFormat(...).build();
```

However, although all code usages of `Pixmap`'s constructor are removed, a couple of problems still remain. Firstly, the original constructor is left unchanged, thus allowing further usages of the it, as opposed to forcing users to call the `PixmapBuilder`. This is problematic because with any new usage another connascence of position is created.

Secondly, as observed from code snippet 6.14, the plugin still detects the usage of `Pixmap` as connascence of position. Worse, it labels its connascence locality as within package, a still relatively dangerous problem. One is thus entitled to think this refactoring has worsen the situation. However, although the connascence of position to `Pixmap` constructor remains, all other usages are replaced with the builder, so in this processed we went down from however many connascences of position there were before, to only one. An elegant solution to this problem is to create the builder within the same class, change the encapsulation of the original constructor to private - essentially enforcing all users to call the builder if an instance is required - and replace its arguments with an instance of the inner builder class.

```
1 public class Pixmap implements Disposable {
2     ...
3     private Pixmap(Builder builder) {
4         this.height = builder.height;
5         this.width = builder.width;
6         this.format = builder.format;
7     }
8
9     public static Builder() {
10        ...
11        // Omitted setters for various fields
12        ...
13        public Pixmap build() {
14            return new Pixmap(this);
15        }
16    }
17    ...
18 }
```

Code 6.15: Refactored constructor of class `Pixmap`

In the above code snippet it is clear that the `Pixmap` constructor has had its connascence of position removed, as it only takes an argument now, and that the only way to create an instance of `Pixmap` is by delegating it to its builder:

```
1 Pixmap.Builder.withHeight(...).withWidth(...).withFormat(...).build();
```

Therefore, to fix the previous refactoring with this, one would need to enforce the builder to reside within the same class as the original constructor with connascence of position, and likewise, change the signature and arguments of the original constructor such that all further usages of the constructor are delegated to the builder.

## 6.2 Performance

One other important aspect of our plugin has always been its performance in terms of how long it actually takes to analyse code. Although at the start we were unsure about how feasible it would be to have the plugin working in real time, that is run its analysis algorithms as every new line of code is written, with time we realised there are series of optimisations that we could use to make this happen, for instance, using multiple threads to analyse the same class, only reanalysing those methods where changes are made etc., as detailed within section 3.1.2.

### 6.2.1 Evaluation method

To evaluate this type of performance, we chose to analyse different codebases and aggregate the results into various indicators. We specifically chose to analyse multiple codebases as we wanted to have sufficiently large testing population, that could encompass all outliers. For this reason, we analysed:

- **libgdx**, since as seen in the previous section it is an extremely large codebase with more than 8 years of development, which covers a lot of potential problems that are often found within gaming platforms.
- **guava** [43], because unlike most projects, this is a Google-written library that is aimed to have long-lasting APIs, thus a big part of the codebase would change less frequently. Given how important frameworks are usually within programmers work, we felt compelled to have a representative example from this category as well.
- **kotlin** [44], because it is a programming language and therefore has a very different underlying infrastructure to most other projects.
- **junit4** [45], because unlike all the other projects, this is a library specifically focused on testing, and so it is reasonable to assume it poses somewhat different programming challenges with respect to our detection algorithms.

Of course, apart from these various codebases there were many others that we have not included. The rationale is that some will have been from the same category with one of the above, but in addition we had to factor in time constraints as well.

To classify our results into meaningful data, we collected performance results on a method basis, rather than classes, as we observed significant discrepancies between codebases in terms of class dimensions, that would have skewed the end results. For example, on average the components from `libgdx` are twice as big as those from `guava`, and slightly shorter than those found in `kotlin`.

Furthermore, to gain statically significance, each individual method was analysed 10 times, and we measured time in nanoseconds for each round of analysis. To make reading the results easier, we present time in milliseconds.

Additionally, please note that this type of evaluation is only possible for the detection analysis component of *Connascer*, since this runs automatically on code. Refactoring capabilities cannot be measured with such an evaluation since, more often than not, user interaction is required both to trigger the refactoring, as well as to give input to various dialog boxes that are required to be completed as part of process. Thus, as the performance characteristics of the refactoring component is relative to each individual user, we shall only present the performance characteristics of plugin's detection component.

### 6.2.2 Evaluation results

	Connasence of Meaning	Connasence of Position
Mean (ms)	3.22	0.029
Standard Deviation (ms)	20.21	0.59
95% interval (ms)	23.43	0.619
Min (ms)	0.0077	0.066
Max (ms)	156.32	47.33

Table 6.1: Time required to analyse an individual method of `libgdx`, `guava`, `kotlin`, `junit4` in ms

The analysis results show that detection of connasence of meaning is slower than its connasence of position counterpart, which is to be expected given the additional workload that goes into the detection for connasence of meaning. However, both detections were on average relatively fast, requiring somewhere in the range of under 3.5ms for inspection of a whole method to be completed in the case of connasence of meaning, and just under 0.03ms in the case of connasence of position. If we consider that our average class, combined over these four codebases, has just over 14 methods, this would put the analysis for a whole class just under 50ms, which we perceive as within the response time required to have it available all the time, even as users add new code.

Furthermore, it should be noted that although there are instances of methods that require somewhere in the range of 150ms for connasence of meaning detection, and 50ms for connasence of position analysis, those are the outliers, rather than the norm. In fact, using the same hypothetical analysis for the average class using the 95% confidence interval time, reveals that a regular class - as detected by average over the four codebases - would be inspected in under 335ms, 95% of the time.

Additionally, it should be also noted that this testing environment does not benefit from caching that our plugin implements. So for example, after analysis of a class is completed the first time, most of the other times we only reinspect those methods where code changes, and only once in a while we inspect the whole class again. Thus, when a user is adding code within say even a large class, maybe well over 20-30 methods, most of the time the analysis cost incurred would be that of the method where code is added, as opposed to the whole class. Thus, the running time for the analysis in real scenarios is greatly improved.

All in all, the performance results were in line with our expectations to provide realtime feedback to users, even without taking into account the optimisations that our plugin benefits from during real programming scenarios. However, the results also showed somewhat relatively larger than

expected analysis time required for the connascence of meaning, thus one of the future goals of the project is to make use of improved algorithms to drive this lower, as close to numbers required by the connascence of position analysis as possible.

### 6.3 Implementation Design

As part of our requirements, we aimed to deliver a solution that was extensible and easy modifiable, to permit further enhancements to be added with ease.

Our end result consists of a relatively modular design, that builds on top of a common infrastructure, the project standard development kit (SDK), that provides common functionality such as code highlighting or detection of source code elements like methods, classes etc.

Code analysis is split into individual components, where each individual component detects specific code problems, like connascence of meaning or connascence of position. One of the benefits to this decentralised approach is that each unit can be thought as running within its own sandboxed environment, and thus an error within any unit does not propagate to the others.

Furthermore, since all analysis units are independent, and as each builds on top of the common project SDK, new analysis units can be further added to the plugin by simply using the SDK, without requiring any knowledge about existing similar analysis units. Our SDK serves, in essence, the same mediator role between interaction with source code and analysis components as an operating system serves for the interaction between hardware and software.

A downside to this modular approach is, however, that performance may be affected. Since all detection units act separately, they cannot communicate with one other. If they had been able to, they could have shared information about different aspects of source code. For example, both connascence of meaning inspection and connascence of position inspection require the connascence locality of certain code elements, such as methods that are used within expressions etc. With our design, both detection algorithms may end up computing the same information twice, as opposed to a design where this information could have been shared between inspections.

Within each individual detection component we have also employed a modular design. That is, detection for example of connascence of position is the aggregation of the different rules that target different code patterns. Based on this approach, any developer who wants to extend existing patterns supported for connascence of position need not understand the whole existing code of this inspection, but can rather focus solely on translating new patterns into code and later add them to the list of associated rules with the inspection for connascence of position. Such flexibility also allows the ability to turn on and off certain rules as desired by user.

A downside to the modular design of detection units may be, once again, the overall system performance. For example, as each detection rule is built following a pattern, and as each detection rule is unaware of the other rules, the same underlying element may be requested multiple times if rules' have overlapping patterns. Depending on how far away in time these overlapping elements are requested, they may add performance overhead to the overall detection analysis. A less modular design could have avoided such cases.

Overall, we appreciate the implementation design to bring benefits that have made some of our requirements possible. However, there are also downsides to our approach that need to be addressed within future work. In addition, a more conclusive evaluation of this aspect is required, especially by other developers, to also include views that are more objective than ours. A good example of a further evaluation is to request external developers to evaluate our implementation design by testing how flexible they find adding additional inspections, or changing existent implementation of inspections.

## 6.4 Usability

We have designed and implemented the plugin with user functionality in mind throughout the whole process, specifically because one of our core goal was to design a solution that could be integrated in programmers' workflows.

To meet our goals, we provided earlier iterations to users for review, as well as opted to collect monitoring data from users that installed our plugin (of course, only if permitted by users) to learn more about how people interacted with the plugin. Our development approach has also involved shorter iterations, where features were made available as soon as they were implemented, and so to date we have released over 25 versions of the plugin.

Additionally, more than 90 developers have provided feedback to our tool. 40 of them have tested and provided feedback during live demonstration sessions, while the rest have used the plugin directly on their codebases. Thanks to all of them, we were able to gain meaningful information that helped improve our solution.

The rest of the the section is divided into users' feedback and plugin monitoring. The former presents feedback received during live demonstration sessions where we presented *Connascer* to developers and exemplified some of its features using various code examples, while the latter reveals details discovered from analytical data collected from developers who installed and used our plugin directly in their IntelliJ IDE.

### 6.4.1 User feedback

Out of the 40 people who gave feedback during live sessions, most of them were gathered from the undergraduate fairs organised by the Department of Computing at Imperial College.

The main goals during these sessions were to present the plugin and its functionality to users, but also to gain insight from having them program with it. The means of obtaining feedback during these sessions was by filling in pre-made feedback forms. As the sessions were spread over the course of 3 weeks, at each stage we put more emphasis on different parts of the application in the feedback forms, however some questions remained throughout all sessions. Furthermore, it should be noted that most of the questions that required the user to answer a question about the likelihood of a particular event, were always based a scale of 1 to 10, with 1 being unlikely, and 10 being highly likely.

#### 6.4.1.1 Plugin Relevance

One of the sections that remained throughout all the different feedback session was on the topic of plugin relevance. Our aim was to learn from users' feedback how relevant they felt that the plugin was with respect to their programming workflow, if they felt that the plugin was focusing on errors they were also likely to commit in their programs etc.

It came with no surprise that people were increasingly likely to agree that the problems reported by the plugin could also part of their codebases, as each new iteration of the plugin was presented. Although initially we scored 6.14 with regards to how likely programmers were to make the same mistakes reported by the plugin, at the last demonstration this came out to 8.36. Part of this improvement is reflected by enhancements that were brought to the detection algorithms between sessions.

Perhaps equally unsurprising is the fact that the likelihood of users to detect problems reported by the plugin went down with each session. Although the initial score of how likely users were able

to detect mistakes without additional aid, that is without our plugin, was 6.76, in the end it came down to 4.87.

These two results were particularly encouraging as it meant the overall relevance of the plugin increased with every new iteration. More over, these results indicated the fact that although more than 8 in 10 programmers were likely to make the errors reported by the plugin, only 5 out of 10 were able to spot these mistakes, thus creating a gap that could be easily bridged by our solution.

One aspect that users thought could do with more improvement was about even more detection algorithms. They thought that since the plugin was good at identifying connascences of meaning and position, it could have been even more useful if it had supported additional connascences. As one of our core goals is to deliver detection of many different kinds of problems that pose a threat to code's velocity to change, we are committed to enhancing current detection algorithms, however due to our limited time available for implementation, further detection is part of our future work.

#### 6.4.1.2 Plugin Usability

The usability of our plugin was measured both through user interactions and their comments about it, as well as through questions on various aspects of the plugin that were adjusted according to which version was used for demonstration.

One of our measures for plugin's usability was based on the interaction with plugin suggested refactorings. Based on their answers, we noted how some refactorings, such as "replace literal with a class constant" scored relatively high on the scale of how likely users were to come up with a similar fix, roughly 8.67, while some other refactorings like "replace constructor with builder" or "wrap method return type with `Optional` type" scored relatively low on the same scale, roughly around 2.8 in both cases.

However, we did not find any correlation between how likely users were to come up with a similar fix and how likely they were to apply that fix. Almost all refactorings scored in range of 7.5-8.5 when users were quizzed about how likely they were to accept/use the proposed refactoring.

Although in general they were likely to use the proposed refactorings, users noted that there were also cases when, depending on the context, they would reject those refactorings. Most of their concerns were when the refactoring ended up changing a lot of classes, for example when the type of a method was refactored, and the method was used in several different parts of the system. They also suggested that for such cases the plugin should integrate with a testing framework that runs tests associated with refactored code before and after a change to verify correctness. Given the limited time available for the implementation and the complexity of such task, we have decided to leave this task as part of future work.

Since our inspection criteria differentiates between instances of connascence with different levels of locality, users also requested some way of disabling inspections for less serious offences, or put another way, to have the possibility to highlight only the most serious errors. Based on this feedback we have added such an option that was successfully tested during the last session of demonstration, and offers the choice between displaying all errors, only those that happen between classes, or only those that happen between package boundaries.

Another very interesting point that was raised by users concerned the colouring scheme used for reporting errors. Although on average 7 out of 10 agreed with the default options, there were also concerns that sometimes they could confuse the red colour used for highlighting the most serious mistakes for a compilation error. Their feedback proposed solutions that either allowed users to change highlighting colour, or used a different colour to highlight such errors. Although attempted under the course of this project, due to the instability of its implementation, we decided to not include in the final iteration, but rather add it to the task of future work.

Lastly, one of the most positive aspects of the demonstration sessions had been the user interest in operating the plugin directly in their IntelliJ IDEA. This was reflected by their likelihood to install the plugin, as measured by our feedback forms, which resulted in a very positive 8.66 likelihood. This means that almost 9 in 10 users would like to try the plugin as part of their development workflow, which considering that one of our goals has been on developing a solution that users would like to have within their workflows, encourages us about plugin's future prospect and our chances of achieving the initial goal.

### 6.4.1.3 Improvements based on users' feedback

On top of specific questions regarding various parts of functionality or background questions, we also received additional comments regarding various aspects that could be improved as well as feedback on why users thought they could or could not benefit from our plugin, additional to what has already been mentioned.

Some of them mentioned that their regular work involved using specific frameworks like `JUnit` for testing, or other frameworks for inverting dependencies which unfortunately our plugin does not support. Thus, they were skeptical of the real benefit they could get from using our tool. Our plan to help increase the satisfaction of users is to enable support for various popular frameworks, so that more and more developers can get to enjoy the comfort of relying on our plugin for detecting potential errors within their codebases.

Others were delighted by the prospect of having a plugin that could do all the checking for them. The most encouraging feedback was that although most of users were aware of some of the problems, if not the majority, unless they were forced to act upon those errors, which they pointed out that our highlighting mechanism did, they felt uncompelled to take any action. This was a great finding, as it further approved our chosen mechanism to deliver feedback about code problems.

Some users also felt that although they could intuitively tell about how various connascences affect the codebase, they needed a quick start guide to provide all necessary details about what the plugin does and how to quickly interact with it. Therefore, based on this feedback we have created an accompanying website that aims to deliver starter information about how users can access all functionality, a legend about what each type of highlighting stands for, and lastly code snippets with explanations why errors occur. This was made available as part of the installation procedure.

One of the other positive remarks users had when they tried the plugin was that they did not perceive that the responsiveness of the overall IDE was affected, and hence they said that they were happy to have it always on, as it could potentially help them correct problems as they are created. Indeed, although not initially part of the core goals, we mentioned before that a performance goal threshold was set as implementation revealed the possibilities of our analysis, and so we were happy to learn that users also perceived the plugin as being responsive for its time required to detect errors.

Lastly, as part of our continuous effort of trying to meet user demands, we have also set up a communication channel where people can report errors about our tool as well as make further suggestions to how our plugin can be improved to meet their expectations.

## 6.4.2 Plugin monitoring

According to the *Hawthorne* effect, people are more likely to lie and to provide favourable answers for the interviewer in surveys [38]. This is one reason why such questionnaires are believed to present skewed data.



Our solution to this problem was to develop a monitoring system within our plugin that aimed to collect unbiased data about how developers used and interacted with it. We recorded data such as the number of times a developer used our provided inspections together with the timestamps when those inspections were triggered, and data regarding whether developers used any of the suggested refactorings and, where applicable, the timestamps when those refactorings were used.

All data was aggregated within a *restdb.io* database [39], in part because access to the database was supported through REST APIs [40], that we took advantage of using network requests, and also because *restdb.io* provides a free tier for data storage with enough functionality for our collection purposes.

We also used an additional small HTTP request library, **Unirest**, to facilitate writing Java access queries to the database [41]. An illustrative example of such query is provided in code section 6.16. Received data was analysed using Python scripts based on the **Pandas** framework.

```
1 public void sendAnalytics() {
2     HttpResponse response = Unirest.post(databaseAddress)
3     .header("content-type", "application/json")
4     .header("x-apikey", databaseKey)
5     .header("cache-control", "no-cache")
6     .field("userID", userID)
7     .field("inspections", inspections) // JSON-formatted string
8     .field("refactorings", refactorings) // JSON-formatted string
9     .asString()
10
11     // handle the HttpResponse
12 }
```

Code 6.16: *POST* request to our collection database

Although we were only able to gather a few days worth of usage, as the plugin monitoring functionality was developed only after our main goals were implemented, we detected some interesting patterns.

First of all, the average user spent just over 52 minutes programming with our solution, as measured by the difference between the first and last timestamp when users sent analytical data, aggregated over all users. This is an encouraging number especially keeping in mind that some of these users were students who were encouraged to test our solution, but who may have been busy with other tasks during the period when we collected data, which unfortunately also overlapped with weekend days.

Secondly, each user applied at least four provided refactorings, which means that on the one hand interaction with the plugin was relative smooth, and on the other hand that our detection algorithms may target problems that are found within many codebases.

Thirdly, although on average there was a good number of interaction with most refactorings, with the average refactoring being used at least nine times, the “wrap method return type with optional” refactoring had only been triggered four times in total. This result reflects that perhaps more emphasis is required on making this type of refactoring as easy to use as the others.

In conclusion, although we were unable to get as much analytical data as we would have wanted, we were encouraged to see a relatively good interaction from developers with *Connascer*, however, as the data revealed, more emphasis should be put on streamlining some of the provided refactorings.

## 6.5 Remarks

With regards to an overall view of plugin's evaluation, we think that our solution is a good initial product that would help programmers identify and correct some of the problems that may affect their codebases. We also realise that this version is not perfect by any means, as we have previously seen, not all refactorings work perfectly, and likewise some of the errors that should have been detected could escape analysis. Additionally, there is also room for improvement in targeting further types of connascences, as well as adding support for various different frameworks.

On the other hand, factoring in the limited time available for researching and developing our plugin, as well as the numerous difficulties encountered with developing our plugin based on the IntelliJ platform, we are content that the user feedback received and presented above has generally been positive.

The generous user input has helped us strengthen our commitment to deliver a solution that works great for everybody, and we have already acted towards this by planning further improvements as well as enabling means of communication with developers, so that future releases can receive more careful testing and further features can be more easily suggested by them and thus incorporated in our planned future work earlier on.

# Chapter 7

## Conclusion

Throughout the course of our project, the main goal was to develop a tool to help programmers visualise problems in their code, as well as to provide automatic refactorings to help remove such problems.

Without claiming that *Connascer* is by any means perfect, and acknowledging that there is room for further improvements, we point out that by the end of the allocated time for this project we have achieved a good scope of the initial goals as it can be inferred from the our overall evaluation.

To go into further details about particular achievements, as well as to talk about future work, lessons learnt and what we would do differently next time, we lay out individual sections below.

### 7.1 Achievements

In the end, *Connascer* has come as far as being used by over 90 developers, and based on the positive feedback that resulted as part of both user interaction as well as other tests, we have identified five key accomplishments of our work.

#### Stable detection plugin

First of all, perhaps one the most important is that the final iteration was delivered in a stable release, that was used by developers directly on their codebases, and which achieves some of the requirements set out initially, more specifically it supports detection of connascence of meaning and connascence of position, and offers users detailed knowledge about the potential impact of each detected problem in the context of their application.

#### Automatic problem refactorings

One of the other achievements of this project has been the implementation of automatic refactorings, that can solve detected problems with minimal user interaction. As presented in the evaluation, not all refactorings work all the time, but we detected a rate of success close to 80%, which is encouraging for an initial implementation.

Furthermore, as measured by user feedback, most of them agreed with suggested refactorings and would apply them in practice. Likewise, we mention that there were also users who questioned our proposed refactorings efficacy in the context of their work, especially if this involved additional frameworks. This is one area which requires future improvements.

## Real-time analysis

Equally important as previous achievements is our solution’s ability to detect problems in real time, as every new line of code is added, or any new software component is inspected. We were particularly happy to achieve this feature especially since users can prevent future problems from persisting in their codebases by correcting them right when they are created.

## Extensible design

Another goal of *Connascer* specifically targeted an extensible design, because we realised that as change is invariant with software, so too will our requirements change sooner or later. Moreover, we wanted to create a modular design that was able to support further detection algorithms with ease, thus following the “software should be closed for modification, but open for addition” mantra [27].

As per the evaluation from section 6.3, we can say that even in the initial release of *Connascer*, this goal has been met from the point of view of being able to add additional detection algorithms without changing any existent code.

## Plugin availability

One very important aspect of the final tool is that it is widely available for anybody who writes their software using IntelliJ IDEA. As some other plugins available through IntelliJ’s plugin market have achieved downloads in the range of millions, for example the *Markdown support* plugin has been downloaded more than 2.2 million times, we are happy that the potential market for developers that can start using our plugin right away is so big.

## 7.2 What we would do differently next time

In addition to the above-mentioned achievements, there are also things that, if we had to start this project again, would do differently because there were also times where we did not choose the best solution.

At the start of our project, we focused on delivering a solution that we hoped worked for everybody, as opposed to a more limited number of users that we could have targeted with an IDE plugin. In the end, the solution proved highly time consuming, because part of developing an IDE plugin is that some functionality, for example code parsing, is already provided, whereas without an IDE we would have been required to implement most of similar tasks ourselves, thus reducing the time available for more important features. As we have already seen, our choice to build on top of the IntelliJ SDK did not prove costly when considering the number of potential developers we can target.

Another aspect we would like to have done differently is the focus on earlier user feedback. Although we have provided users with various iterations, we could have incorporated even more of their demands had we focused on this earlier on. Likewise, establishing an earlier communication channel with users would have also helped, as we would have learnt sooner about some of their requirements such as integration with various frameworks.

At last, next time we would like to add more tests to our plugin, to capture those cases of undetected connascences or failed refactorings.

### 7.3 What we have learnt

First and foremost, one of the most important lesson learnt is time management and task allocation in a small team, as opposed to a normal-sized team that we have been used with from previous experiences.

If in bigger teams task distribution and allocation is more permissive, there is little room to accustom these in a small team. This can be challenging especially when we are required to do things that in other teams were delegated to others, such as front-end development or proof reading, and becomes increasingly important when working towards a deadline, as some tasks can take longer than expected.

Another key aspect that we have learnt is related to the value of users input in development. We learnt how user feedback can literally change priorities within the development pipeline, or it can provide details that were initially left out. We learnt that having as much user feedback is key, especially when the end product is expected to be used by many people, and asking for it as early as possible is just what developers behind such products should do.

Finally, we also learnt how we should not make underlying assumptions about whether a piece of functionality is going to be obvious to users or not. We were surprised that some users felt confused to how some of the earlier iterations of *Connascer*, especially as we thought that most of the functionality will not require any explanation. We could not have been more wrong about it, but our interaction with users helped reveal and correct these issues.

### 7.4 Future work

All in all, the initial accomplishments of *Connascer* are satisfactory, however, as revealed by both our evaluation as well as user input, there is room for future improvements. In addition, we know that there is no such thing as the perfect software, since there is always something more that can be added or improved. Therefore, we are committed to enhance our solution to attract even more users, and so we have established a roadmap of features we want to add.

Amongst the most important features is the requirement to provide more stability to the plugin, essentially aiming to fix all problems that were revealed by the evaluation, namely those instances that have not been detected when they should have been, and those refactorings that failed to solve the initial problem.

As one of the most requested feature from our users, we also plan to add a customisable colour palette for error highlighting. This would enable those with special needs to adjust the default scheme to their preference, as well the others decide what works best for them.

Integration with various other frameworks, such as the popular `JUnit` testing framework is also something we put up on the list of features we want to add support for, as we realised that a substantial amount of users requested something like this.

As part of improving our detection algorithms, we want to add support for the degree property of connascences. Currently, we only consider strength and locality, but disregard degree whenever we present possible errors to users.

Finally, we plan to add future support for detection of more connascences, such as connascence of execution order. Additionally, support for a reliable detection of islands of cohesion is required, since, as mentioned in our proposed solution analysis section 2.4.7, this can complement the weaknesses which connascence suffers from within the detection of problems inside a class. This is perhaps the most challenging aspect of our future work, but thanks to the experience gained so far and to our existing infrastructure we are convinced we can achieve it.

# Bibliography

- [1] Wikipedia. Subroutine. Available at: <https://en.wikipedia.org/wiki/Subroutine>
- [2] Wikipedia. Static Program Analysis. Available at: [https://en.wikipedia.org/wiki/Static\\_program\\_analysis](https://en.wikipedia.org/wiki/Static_program_analysis)
- [3] M. Lanza and R. Marinescu. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. *Springer Publishing Company, 2010*
- [4] M. Page-Jone. Fundamentals if Object-Oriented Design in UML. *Addison-Wesley, 2000*
- [5] Collective authors. What is connascence?. Available at: <http://connascence.io>
- [6] S. Metz. Practical Object-Oriented Design in Ruby: An Agile Primer. *Addison-Wesley, 2012*
- [7] B. Foote and J. Yoder. Big ball of mud. *Pattern languages of program design, 1997* Available at: <http://joeyoder.com/PDFs/mud.pdf>
- [8] J. B. Rainsberger. The Three Values of Software. Available at: <http://blog.jbrains.ca/permalink/the-three-values-of-software>
- [9] M. Fowler, K. Beck, et. al. Refactoring: Improving the Design of Existing Code. *Object Technology Series, Addison-Wesley*
- [10] Spa Conference. Available at: <http://spaconference.org/spa2017/>
- [11] Unknown authors. JArchitect: How to improve your Java code quality. Available at: <http://www.jarchitect.com/>
- [12] Unknown authors. Checkstyle: Java static analysis tool. Available at: <http://checkstyle.sourceforge.net>
- [13] Unknown authors. Cobertura: A code coverage utility for Java. Available at: <http://cobertura.github.io/cobertura/>
- [14] Unknown authors. Valgrind: instrumentation framework for building dynamic analysis tools. Available at: <http://valgrind.org>
- [15] A. Hicken. False Positives in Static Code Analysis. Available at: <https://blog.parasoft.com/bid/113524/False-Positives-in-Static-Code-Analysis>
- [16] Unknown authors. List of tools for static code analysis. Available: [https://en.wikipedia.org/wiki/List\\_of\\_tools\\_for\\_static\\_code\\_analysis](https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis)

- [17] A. Hunt and D. Thomas. The Pragmatic Programmer. *Addison-Wesley, 2000*
- [18] Wikipedia. Broken windows theory. Available at: [https://en.wikipedia.org/wiki/Broken\\_windows\\_theory](https://en.wikipedia.org/wiki/Broken_windows_theory)
- [19] JetBrains. IntelliJ Idea. Available at: <https://www.jetbrains.com/idea/>
- [20] Checkstyle. <http://checkstyle.sourceforge.net>
- [21] Find Bugs. Available at: <http://findbugs.sourceforge.net>
- [22] PMD. Available at: <https://pmd.github.io>
- [23] Wikipedia. Cyclomatic Complexity. Available at: [https://en.wikipedia.org/wiki/Cyclomatic\\_complexity](https://en.wikipedia.org/wiki/Cyclomatic_complexity)
- [24] N E Fenton, M Neil (1999). A Critique of Software Defect Prediction Models. *IEEE Transactions of Software Engineering*, 25 (3): 675-689
- [25] L Hatton. The role of empiricism in improving the reliability of future software. *Windows, 2008*
- [26] Wikipedia. Unified Modeling Language. Available at: [https://en.wikipedia.org/wiki/Unified\\_Modeling\\_Language](https://en.wikipedia.org/wiki/Unified_Modeling_Language)
- [27] R. C. Martin. Agile Software Development, Principles, Patterns, and Practices. *Pearson, Second Edition, 2002*
- [28] Eclipse. Available at: <https://www.eclipse.org>
- [29] Coverity. Available at: <http://www.coverity.com>
- [30] JArchitect. Available at: <http://www.jarchitect.com>
- [31] JTest. Available at: <https://www.parasoft.com/product/jtest/>
- [32] Squale. Available at: <http://www.squale.org>
- [33] ER Relationships. Available at: [https://en.wikipedia.org/wiki/Entityrelationship\\_model](https://en.wikipedia.org/wiki/Entityrelationship_model)
- [34] A. Aho, J. Ullman, M. S. Lam, R. Sethi. Compilers: Principles, Techniques, and Tools. *Pearson Education, Second Edition, 2006*
- [35] R. C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. *Prentice Hall, 2008*
- [36] IntelliJ Psi Documentation. Available at: [http://www.jetbrains.org/intellij/sdk/docs/basics/architectural\\_overview/psi\\_elements.html](http://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi_elements.html)
- [37] Null Object Pattern. Available at: [https://en.wikipedia.org/wiki/Null\\_Object\\_pattern](https://en.wikipedia.org/wiki/Null_Object_pattern)
- [38] Hawthorne Effect. Available at: <https://explorable.com/hawthorne-effect>
- [39] RestDB. <https://restdb.io>

- [40] M. Masse. REST API Design Rulebook. *O'Reilly*, 2011
- [41] Unirest Java Framework. Available at: <http://unirest.io/java.html>
- [42] LibGDX Framework. Available at: <https://github.com/libgdx/libgdx>
- [43] Guava Framework. Available at: <https://github.com/google/guava>
- [44] Kotlin Programming language. Available at: <https://github.com/JetBrains/kotlin>
- [45] JUnit4 Framework. Available at: <https://github.com/junit-team/junit4>