ConCAD: A Tool for Interactive Detection of Code Anomalies

Danyllo Albuquerque
Federal University of Campina Grande (UFCG)
Campina Grande, Paraíba, Brazil
danyllo@copin.ufcg.edu.br

Everton Guimaraes
The Pennsylvania State University
Malvern, Pennsylvania, United States
ezt157@psu.edu

Mirko Perkusich
Federal University of Campina Grande (UFCG)
Campina Grande, Paraiba, Brazil
mirko@virtus.ufcg.edu.br

Hyggo Almeida
Federal University of Campina Grande (UFCG)
Campina Grande, Paraiba, Brazil
hyggo@virtus.ufcg.edu.br

Angelo Perkusich
Federal University of Campina Grande (UFCG)
Campina Grande, Paraiba, Brazil
perkusich@virtus.ufcg.edu.br

ABSTRACT

Code anomalies are indicators of software design can potentially decrease software maintainability and they are associated with an explicit set of refactoring actions. However, Detection of code anomalies is traditionally supported by Non-Interactive Detection (NID) techniques. These techniques encourage developers to reveal anomalies in later revisions or versions of a program, implying in counter-productive or even prohibitive refactoring actions. In this context we created ConCAD as an eclipse plug-in that enable Interactive Detection (ID) of code anomalies. This tool provide developers' support to reveal anomalies when code fragments are still being edited, encouraging early and continuous detection of code anomalies.

KEYWORDS

Code Anomalies, Tool, Software Quality; Refactoring

1 CONTEXTUALIZATION

Code anomalies have been proposed as a way for developers to recognize the need to restructure their software [3]. Because code anomalies can go unnoticed while developers are working, techniques for anomaly detection have been developed to alert programmers and to help them understand the cause of those anomalies. These techniques are comprised of two components [5][1]: 1) *Detection Mechanism* which allows developers to define algorithms and choose some metrics - and adjust thresholds - to compose the detection strategy [4]; and 2) *User interface* which displays the results of anomaly detection.

There are two main ways of presenting the detection results: (i) list-based and (ii) interactive-based. The first represents a list of code anomalies throughout the project, while the second uses the source code to alert potential instances of code anomalies. Based on the developer's interaction with the aforementioned components, the techniques can be classified in two ways.

Interactive Detection (ID) technique supports the developer's interaction with code anomalies by revealing instances in code fragments without an explicit developer's request. The ID techniques continuously work in the background to detect anomaly instances while software developers work on coding tasks. A developer using ID techniques can identify anomaly instances earlier, allowing developers to analyze and modify the source code while interacting with the affected code elements [1][5]. **Non-Interactive Detection (NID)** technique does not support the developer's interaction with affected code elements unless the detection mechanism receives the developer's request, after which it will detect potential code anomalies by analyzing the entire project. Developers using NID techniques may identify anomaly occurrences only after the code churns are merged with other software components. Finally, once developers directly interact with the detection mechanism and a global list of anomaly instances, they cannot concurrently perform other programming activities [1][5].

Most studies on the detection of code anomalies strictly focused on evaluating of NID [6][9][2]. These studies pointed out that NID techniques induce a low number of correctly detected code anomalies, directly impacting their effectiveness. Other studies suggested that NID techniques induce ineffective refactoring actions [7, 8]. We assume that ID techniques may promote the early identification of code anomalies by allowing developers to perform effective refactoring actions. Although the ID technique seems promising for detecting code anomalies and identifying refactoring opportunities, there is a lack of empirical knowledge regarding its effectiveness in these activities. This observation raises the need for tooling support that enables ID characteristics.

To address this research problem, this study presents a tool called ConCAD (Continuous Code Anomaly Detection) that supports identifying and prioritizing 10 different types of code anomalies. The main benefit of using ConCAD is that developers can configure and extend the tool by providing different strategies to identify and prioritize code anomalies. Aims to preliminary validate the ConCAD tool, we conducted a controlled experiment involving 16 subjects among graduate students and developers. These subjects performed tasks related to detecting code anomalies, supported by both ID and NID techniques. The controlled experiment allowed us to investigate the ID technique's influence in performing the abovementioned tasks.

2 MOTIVATING EXAMPLE

We start our motivating example with a software developer called Tom. He is working on the Apache Tomcat®project, and he experienced difficulty in adding functionality to the *JNDIRealm*[1] class.

[1]https://tomcat.apache.org/tomcat-8.5-doc/api/org/apache/catalina/realm/JNDIRealm.html

This class contains eight methods like the *compareCredentials*, illustrated in Figure 1. In this case, the code anomaly he noticed is called *Data Clumps*. It is observed when the same group of data objects (i.e., context and credentials - See red rectangle in Figure 1) is used in several different places. Data clumps can make the software more difficult to maintain because if the representation of one of the data objects changes or the protocol for manipulating those objects changes, then every location in which the group of objects appears must be examined to see if it needs to be modified.

```
protected boolean compareCredentials(DirContext context,
    User info, String credentials) throws NamingException {
    . . .
    /* sync since super.digest() does this same thing */
    synchronized (this) {
        password = password.substring(5);
        md.reset();
        md.update(credentials.getBytes());
        String digestedPassword =
            new String(Base64.encode(md.digest()));
        validated = password.equals(digestedPassword);
    }
```

Figure 1: Code fragment of the *JNDIRealm* class.

Assuming that Tom was using a NID technique, the Data Clumps would be detected at a late stage when many modules already depend on it (creating more coupling). Thus, refactoring actions can be performed with more effort or might even be impossible to be performed by Tom. In turn, if Tom uses the ID technique, he could inspect the class and opportunely conclude that the *context* and *credentials* parameters should be encapsulated into a single object by performing extract class refactoring because these parameters had appeared together in the parameter list of several different methods depending on the *JNDIRealm* class. Thus, the ID technique allows detection of code anomalies earlier, which may lead to less effort to perform refactoring than traditional techniques.

3 CONCAD TOOL

This section introduces the ConCAD tool in detail. Initially, an overview of the tool is presented, highlighting its main components and characteristics. Then, details regarding the architecture as well as the implementation of the tool are described. Finally, the main features of this tool are provided.

3.1 Main Features

ConCAD takes the Java source code as input and produces a display of the results associated with detecting code anomalies. Basically, there are two main profiles for using the tool: developer and user. The first is directly related to the tool's construction (or expansion), while the second is directly related to the use of its features. Once the ConCAD tool is instantiated in the integrated development environment, the user can interact with the tool to do one or more of the following activities:

- **Mapping Information**. The user can include external information in ConCAD through mappings to the source code. For example, the system's architectural design can be mapped to the code that aims to indicate which classes are directly related to each architectural component.
- **Anomaly Detection**. The source code of the developer's current project is analyzed through several relevant metrics in the context of Object Oriented Programming (OOP), and code anomalies are automatically detected through a combination of detection strategies.
- **Anomaly Prioritization**: Code anomalies are evaluated according to their criticality and severity based on different prioritization criteria. These criteria can be easily customized by users according to their preferences or even according to project characteristics.
- **Choose Detection Approach**. ConCAD is designed to perform code anomaly detection according to interactive and non-interactive techniques. The developer can choose one of the strategies according to their suitability for his programming activity.
- **Visualization of results**. Since the ConCAD tool was designed to support different detection techniques, ways of visualizing the results must adhere to such approaches. While in the ID technique, more local results are required, in the case of the NID technique, there is a need to display more global results.

The ConCAD tool supports the calculation of 30 different types of OOP metrics. These metrics are used to compose detection strategies for different types of anomalies. A detection strategy consists of a quantifiable and logical expression where code fragments that do not conform to the rule are detected. It is essential to mention that ConCAD provides support for configuring the anomaly detection strategy based on user preferences that may be previously stored in the database. Based on OOP metrics and pre-configured detection strategies, the ConCAD tool provides initial support for detecting 10 different types of code anomalies: *Brain Class, Brain Method, Feature Envy, Data Class, Dispersed Coupling, God Class, Intensive Coupling, Refused Bequest, Tradition Breaker*, and *Shotgun Surgery*. It is important to note that based on the various metrics already implemented in the tool, new and different detection strategies can be easily configured to expand the anomalies supported by the tool.

The number of code anomalies can pose a great difficulty when analyzing large systems. Therefore, ConCAD provides a means to prioritize each instance of the 10 distinct types of anomalies. From its inception, this tool was designed to be flexible enough to strike a balance between (i) prioritization criteria that require little user intervention but work better after multiple versions of the system and (ii) criteria that rely on (a fair amount of) external information provided in advance by the user, but produce good results for recent system versions. This allows users to prioritize inspection and correction of the most critical fragments according to their preferences.

3.2 Architectural Details

The ConCAD tool was designed according to six main components described in Figure 2.

Eclipse Platform. Being an Eclipse *plug-in*, ConCAD has a direct dependency on the Eclipse Platform core, more precisely on the following three components: *workspace*, *workbench* and the *incremental project build*. The *workspace* is used to gain access to the resources of Eclipse projects and their interdependencies. The

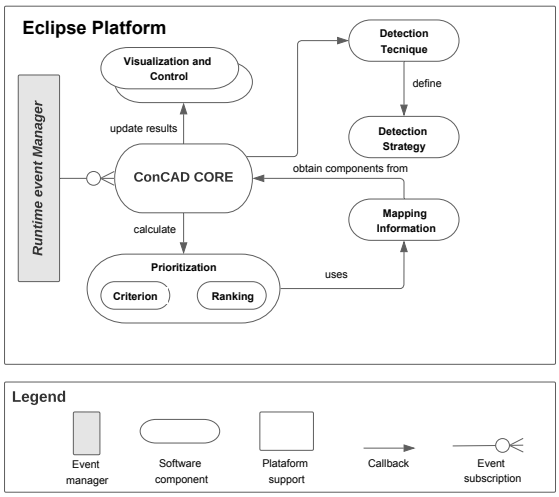

Figure 2: ConCAD Architecture.

workbench is used to define the markers that the tool will use to annotate the source code files with the anomalies detected in the project and to define the visualization forms that ConCAD needs to present its more localized analysis results. Finally, *incremental project build* mechanism is used to get the code elements (e.g., *statements*, methods, classes, and packages) that have changed since the last build and also the change factor that describes the actual changes on a certain element.

ConCAD tool uses the Eclipse JDT project [2] (*Java Development Tools*) aims to provide the necessary means for developing *plug-ins* to extend the functionality of the Eclipse. Thus, ConCAD needs information about the code elements involved in its various metrics, detection strategies, and means of visualization and exploration. Most of this information is provided by the *Java Model* which represents a Java project, providing a lighter model associated with code elements. However, more detailed information such as cross-references (e.g., method calls and variable accesses) and low-level program elements (e.g., parameters, local variables, and return types) are only available through AST-based objects. It is important to mention that most of the ConCAD analyses depend on this information based on the *Java Model* and AST models.

ConCAD Core. This component is responsible for managing the loading and analysis of source code files in Java through its direct interaction with the Eclipse platform. Thus, this component is concerned with obtaining and treating information based on *Java Model* and AST to provide the tool's functionality. In general terms, it manages the tool's workflow by detecting code anomalies and interacting with the prioritization component to generate the classification of detected anomalies.

Detection Strategies. This component implements various strategies for detecting each type of code anomaly. This component uses threshold values for OOP metrics previously configured or obtained from users through visualization and control mechanisms in detection strategies. Furthermore, new and different detection strategies can be easily added by developers through this component.

Mapping Data. This component supports the framework for mapping external information to the source code. The user can also configure new types of mappings to be used by prioritization strategies. One way to take advantage of external information when anomalies are analyzed and prioritized is by mapping elements of the *design* model (e.g., modules, responsibilities, and scenarios) to the implemented code elements (e.g., packages, classes, methods). In this way, ConCAD can be instantiated by a developer to take into account different types of architectural requirements and documentation, such as component and connector views, scenarios based on quality attributes, among others. In all cases, the tool user will map an element of the *design* model to one or more classes (or packages) to indicate its traceability.

Prioritization Criteria. This component calculates a rating for a set of anomalies detected by the tool. How the calculation is carried out, as well as the application of the criteria, are extensible. That is, new prioritization criteria can be added by developers without changing the architecture, thanks to a set of interfaces. A user can instantiate ConCAD to prioritize anomalies by different criteria (e.g., the relevance of the anomaly type, system version history, and different software metrics). In addition, the user can use external information to improve prioritization.

Visualization and Control. This component represents a set of user interfaces (e.g. views, dialog boxes, and action handlers) that aim to provide the tool's functionality to its users. In this way, the user can interact with the user interface to map external information to the source code, define their detection strategies, define their prioritization criteria, and analyze the results associated with detecting code anomalies.

3.3 ConCAD Operation

This section demonstrates how ConCAD Eclipse works. In what follows, we explain (i) how anomalies are detected and exposed, (ii) how detection strategies are configured, and (iii) the ways of prioritizing detected anomalies.

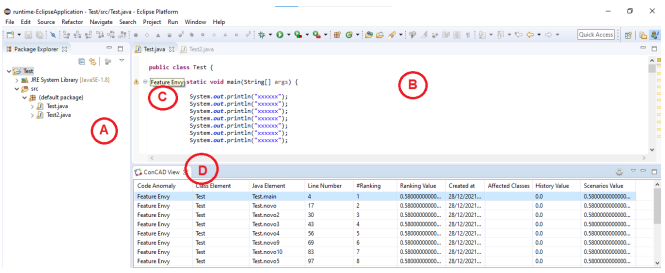

Figure 3: ConCAD overview.

Iteractive Detection. After a project is loaded into the Eclipse environment, the user can instruct ConCAD to identify and analyze possible code anomalies in a given snippet. This is done by clicking on the button "ConCAD Detector -> Enable ConCAD" in a context menu in the project (Figure 3.A). After clicking on this option, the user switches their context to perform some programming activity

[2] https://www.eclipse.org/jdt/overview.php

on a particular code file. On this occasion, the user needs to focus his programming activities on code fragments that are a constituent part of the file active in the Integrated Development Environment (Figure 3.B).

The user performs some programming activity on the code fragment. At this point, the anomaly detection mechanism acts - regardless of an explicit request from this user - calculating metrics associated with the fragment and related source code files. This mechanism performs a detailed analysis of the metric values and, if there is a violation in the previously defined detection strategies, the developer is alerted about anomalies through markers in the considered code fragment (Figure 3.C). Finally, the developer can also have a more global view of the results associated with code anomalies present in the project. All these occurrences are listed and classified in *ConCAD View* (Figure 3.D).

It is important to note that ConCAD tool markers are dynamic: as new code is written or modified, new markers may appear, or existing ones may disappear. Although anomalies are detected using strategies, the problem is described in terms of design concepts rather than numbers. In this sense, ConCAD lists the actual entities and relationships that contribute to a specific code anomaly, helping the developer to identify and remove the anomalies correctly.

Non-Iteractive Detection. Similarly, by clicking on the button "ConCAD Detector -> Find Code anomalies" in a context menu in the project (Figure 3.A), the tool will analyze the metrics associated with all the code elements of the project in question. As soon as the detection engine identifies the anomalies present in the project, these occurrences will be listed and classified in the *ConCAD View* (Figure 3.D).

The user switches his context to perform some programming activity on a particular code file. On this occasion, the user needs to focus his programming activities on code fragments that are a constituent part of the code file that is active in the Integrated Development Environment (Figure 3.B). Upon completion of the programming activity, the user needs to explicitly request the tool so that the detection engine performs its analysis in search of code anomalies. More importantly, in this operating mode the user is deprived of localized anomaly detection results (e.g., markers in the code). As a consequence, new occurrences of anomalies can be inserted and be detected only late when many modules depend on or are affected by these anomalous sections.

Detection Configuration. The ConCAD tool allows the selection of different thresholds for the metrics used on anomaly detection. In Figure 4 we can identify the detection strategy for the anomaly classified as *God Class*. This anomaly has a strategy based on three distinct metrics (i.e., ATFD, WMC, and TCC) together with their threshold values.

While the developer who instantiates the tool can pre-configure the thresholds by default, the tool's user can change the thresholds to adapt detection strategies to the characteristics of the analyzed system. It is worth noting that each of the 10 distinct anomalies has similarly adaptable and configurable detection strategies.

Detection Prioritizing. Once code anomalies are detected, they must be classified according to their importance. A developer can instantiate ConCAD to prioritize code anomalies by different criteria. The first strategy concerns the relevance of the code anomaly type (Figure 5.A). The user can assign values (between 1 and 5) to

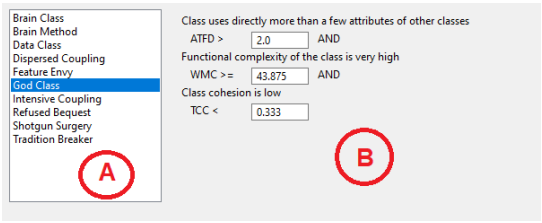

Figure 4: Detection Strategy Definition.

establish a level of importance for each type of anomaly supported by ConCAD. In Figure 5.B, we can see that the user can easily assign importance values to each of the types. These values are sorted later, indicating this user's preference.

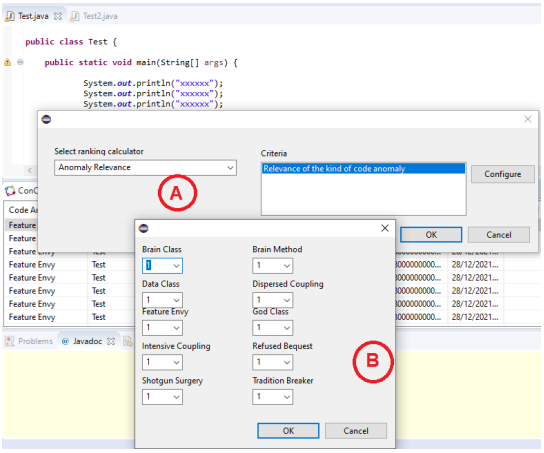

Figure 5: Prioritizing for Anomaly Type.

Another prioritization criterion is associated with the modification scenario. For example, some work has shown that by refactoring code anomalies related to architectural issues, architecture degradation could be stopped [**?**], [**?**]. In this case, the developer can create a criterion first to classify those anomalies that compromise the system architecture in a given modification scenario. For example, this analysis can be performed by defining a criterion based on the relationship between code anomalies and system modification scenarios. So the external information built into ConCAD becomes essential to determine the criticality of each instance of code anomalies.

4 CONCAD PRELIMINARY VALIDATION

We performed a controlled experiment aims to analyze and compare the effectiveness of code anomaly detection using ID and NID techniques supported by ConCAD. We recruited 8 graduate students and 8 professional developers. These subjects performed the experimental tasks by manipulating Java code files extracted from the "Java Core Library". They were subjected to "two-treatment factorial crossover design" [**?**] for the tasks of detecting code anomalies.

To have confidence in the experiment results, it was necessary to have an "oracle" representing the list of code anomalies that

actually represent maintainability problems. The environment for executing the experimental tasks, including the files and tooling support, was already provided to the subjects. Moreover, each experimental task was individually conducted with the first author as an "observer" and another author as an "auxiliary". The subjects performed code analysis in order to detect eight different instances of code anomalies supported by ConCAD. For doing so, they analyzed two code files: One with ID support and the other with NID support. Table 1 describes the results for all subjects considering: Detected Code anomalies (DCS), True Positives (TP), False Positives (FP), and False Negatives (FN).

Table 1: Results of detection of code anomalies.

	NID				ID		
	TP	**FP**	**FN**		**TP**	**FP**	**FN**
Developer 1	5	1	17	Developer 1	8	1	14
Developer 2	7	2	15	Developer 2	16	2	6
Developer 3	10	1	12	Developer 3	16	3	6
Developer 4	6	2	16	Developer 4	10	2	12
Developer 5	10	2	12	Developer 5	13	1	9
Developer 6	11	3	11	Developer 6	14	3	8
Developer 7	7	1	15	Developer 7	9	2	13
Developer 8	8	2	14	Developer 8	13	1	9
Student 1	5	1	17	Student 1	11	2	11
Student 2	5	1	17	Student 2	9	2	13
Student 3	8	2	14	Student 3	12	3	10
Student 4	6	1	16	Student 4	10	1	12
Student 5	6	3	16	Student 5	9	2	13
Student 6	5	2	17	Student 6	9	3	13
Student 7	6	2	16	Student 7	9	1	13
Student 8	5	3	17	Student 8	10	1	12

ID increases DCS and TP. Regarding DCS, we are interested in knowing how many anomaly instances the developer could detect, independently if the code anomaly instance represents a maintainability problem (i.e., TP) or not (i.e., FP). Regarding results associated with TP, we can notice a similar increment. We identified subjects using the NID to detect 110 TP, whereas by using the ID, they detected 178 TP. Therefore, the use of ID increased by about 60% of the total of TP when detecting code anomalies.

ID decreases FN. Related to FN, we are interested in knowing how many anomaly instances that represent maintainability problems (i.e., TP) developers have failed to detect. The experimental results revealed that subjects identified 242 FN when using the NID, while subjects using the ID managed to reduce the FN number by 28%, totaling 174 FN. The use of the ID led subjects to identify the most anomaly instances that represented maintainability problems. Although we noticed an increase in FP, there are still opportunities for improvement in the detection mechanism.

ID increases precision. The average precision with the ID was 0.86, while with the NID was 0.78. The difference in precision, however, was not significant considering a variation of about 11%. We also observed the subjects' working experience directly affected the precision results. Although the use of the ID increases the number of TP, it also tends to increase FP, directly impacting precision values since these effectiveness measurement components are used to calculate the precision.

ID increases recall. Based on the outcomes associated with FN and TP, it is expected that the use of the ID technique significantly contributes to improving the recall to detect code anomalies. On average, we observed that the subjects using the ID achieved a recall of 0.51, whereas using NID achieved 0.31, representing a difference of about 60% in favor of the ID technique. When analyzing different samples, the developers improve recall values by 50%, while the students improve recall values by about 70%.

It is worth mentioning the results indicated no negative impact on the use of ID technique. The subjects are likely to benefit from detecting anomalies earlier when they regularly receive feedback. Moreover, the constant availability and higher amount of information through ID led subjects to accept a higher number of code anomaly suggested by the technique. More experienced developers using ID obtained fewer FPs than the students (with less working experience) using the same technique. Similarly, developers identified a higher number of TP compared to students.

5 FINAL CONSIDERATIONS

In this paper, we propose a novel anomaly detector called ConCAD that provides an interactive ambient designed to first give programmers a quick, high-level overview of the anomalies in their code, and then, if they wish, to help in understanding the sources of those code anomalies. This tool allows the user to focus on those anomalies that are critical for the system, thus making their analysis cost-effective. The design of ConCAD supports a prioritization schema based on multiple criteria, which can be easily provided and extended by developers. We argue that ConCAD is lightweight because it can already work with a small set of criteria and provide a useful ranking to the users. As future work, we will extend our tool to support refactoring. Our intention is to suggest refactoring strategies for each kind of anomalies. This will help novice developers to analyze refactoring strategies for complex anomalies. Also, we plan to create visualizations to awareness developers of the components of the application most affected by anomalies.

REFERENCES

[1] Danyllo Albuquerque, Alessandro Garcia, Roberto Oliveira, and Willian Oizumi. 2014. Deteccao interativa de anomalias de codigo: Um estudo experimental. In *Proceedings of Workshop on Software Modularity*. sn.

[2] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21, 3 (2016), 1143–1191.

[3] Martin Fowler. 2018. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[4] Mika V Mantyla. 2005. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In *Proceedings of the International Symposium on Empirical Software Engineering*. IEEE, 10–pp.

[5] Emerson Murphy-Hill, Titus Barik, and Andrew P Black. 2013. Interactive ambient visualizations for soft advice. *Information Visualization* 12, 2 (2013), 107–132.

[6] Luciano Sampaio and Alessandro Garcia. 2016. Exploring context-sensitive data flow analysis for early vulnerability detection. *Journal of Systems and Software* 113 (2016), 337–361.

[7] Markus Schnappinger, Mohd Hafeez Osman, Alexander Pretschner, Markus Pizka, and Arnaud Fietzke. 2018. Software quality assessment in practice: a hypothesis-driven framework. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–6.

[8] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of GitHub contributors. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 858–870.

[9] Chris Simons, Jeremy Singer, and David R White. 2015. Search-based refactoring: Metrics are not enough. In *Proceedings of the International Symposium on Search-Based Software Engineering*. Springer, 47–61.