Live Code Smell Detection of Data Clumps in an Integrated Development Environment

Nils Baumgartner, Firas Adleh and Elke Pulvermüller
Research Group Software Engineering, Institute of Computer Science, Department of Mathematics and Computer Science, University of Osnabrück, Osnabrueck, Germany
{*nils.baumgartner, fiadleh, elke.pulvermueller*}*@uni-osnabrueck.de*

Keywords: Code Smell, Data Clumps, Refactoring, Integrated Development Environment

Abstract: Code smells in software systems create maintenance and extension challenges for developers. While many tools detect code smells, few provide refactoring suggestions. Some of the tools support live detection in an integrated development environment. We present a tool for the live detection of *data clumps* in Java with generated suggestions and semi-automatic refactoring. To achieve this, our research examines projects and their associated abstract syntax trees and analyzes types of variables. Thereby, we aim to detect *data clumps*, a type of code smells, and generate suggestions to counteract them. We implemented our approach to live *data clumps* detection as an *IntelliJ* integrated development environment application plugin. The live detection achieved a median of less than 0.5 s for the *ArgoUML* software project, which we analyzed as an example. From over 1500 investigated files, our approach detected 125 files with *data clumps* and that of *CBSD* (Code Bad Smell Detector) detected 97 files with *data clumps*. For both approaches, 92 of the files found were the same. We combined the manual steps for refactoring, resulting in a semi-automatic elimination of *data clumps*.

1 INTRODUCTION

Expenses for the continuing development and maintenance of software projects are not negligible aspects and must be considered during the planning of such projects. Maintaining software may account for between 40 % and 75 % of the total costs, according to (Brown et al., 1998). For software development, these costs include the time for employees to learn a new library or the structure of an existing project. However, a means of lowering the cost of maintenance may already exist during the setup of a software project via flexible and future-oriented structures and a clean code. One such way to maintain software is refactoring, which improves the software quality and, thus, its maintainability without affecting its recognizable behavior (Becker et al., 1999). The exact criteria to identify a clean and good source code are various and not sharply defined. In contrast, recognition of a bad source code is easier and is often associated with the terms "anti-pattern" and "code smell," the latter coined by Kent Beck and adopted by Martin Fowler (Becker et al., 1999). Code smells are places in the implementation that point to potential errors or maintenance problems.

A well-known type of code smell is *data clumps*. According to (Lacerda et al., 2020), *data clumps* are among the top 10 code smells and are the second most common in the web domain (Delchev and Harun, 2015). Thus, they present a problem in software projects that should not be underestimated. *Data clumps* are a group of variables that appear together in different areas in a source code and that point to a possible new data structure. Due to the distribution of *data clumps* across a software project, detection is difficult for a developer.

Integrated software development environments (IDEs) for software provide useful tools for a developer and facilitate the work on software. Support for the automatic, timely and early detection of *data clumps* within an IDE is an important issue for software development and the associated development costs. Already, (Simon et al., 2001), (Gronback, 2003), (Salehie et al., 2006), and (Habra and Lopez Martin, 2006) have employed metrics to detect code smells. In addition to detection, an equally important point is refactoring the *data clumps*, for which manual implementation can be time-consuming and monotonous. A useful effort, therefore, would be an automatic or semi-automatic tool for this.

In this study, we approach a means of detection and support for eliminating *data clumps* integrated in an IDE to ease the developer's work. Furthermore, we demonstrate that the detection of *data clumps* can be sufficiently fast so that live code smell detection is possible. Finally, we evaluate how our approach performs against comparable tools for detection.

The remainder of this paper is organized as follows. Section 2 provides an overview and background of the method presented in this study. Section 3 discusses related approaches, and Section 4 describes our proposed method for detecting and refactoring *data clumps*. In Section 5, we present our evaluation and results, which are discussed in Section 6, along with the challenges and limitations of our approach. Finally, Section 7 presents closing remarks and outlines recommendations for future work.

2 BACKGROUND

This section provides the background to our approach. To develop a form of live code smell detection, we first focus on the term code smell in Section 2.1. Then in Section 2.2, we provide a definition of *data clumps*, followed by the procedure for refactoring them in Section 2.3. Subsequently, the abstract syntax tree (AST), a representation of a source code, is explained in Section 2.4, followed in Section 2.5 by a definition of a program structure interface (PSI) built on top of the AST.

2.1 Code Smell

A code smell is not necessarily a bug but is often an indicator of a deeper problem (Fowler, 2019). Structures that must be improved can be identified by code smells. Various domains can have specific code smells and, therefore, different impacts on the areas of architecture, database, design and implementation (Sharma and Spinellis, 2017). Poor design as a symptom of code smells poses a potential risk for future bugs and loss of code structure. Additionally, it has a negative impact on code in terms of understandability, testability, extensibility and reusability (Fowler, 2019). As a result, code smells can have a negative impact on maintainability. A smell can enter a project in various ways, such as inattention, knowledge gaps, a change in requirements, chosen technologies and frameworks, work processes, organizational structure, team culture or poor resource planning (Sharma and Spinellis, 2017). Each of these factors can influence a project and increase its costs. To avoid a reduction in quality, refactoring may be applied to a software project (Becker et al., 1999). Fowler designates no fixed time for the refactoring but states that it may be included in the workflow.

Different sources (Zhang et al., 2008) and (Fowler, 2019) provide various definitions and quantities of code smells, which result from the subjective definition of a particular code smell (Mäntylä and Lassenius, 2006). Therefore, a complete list of code smells is not available. A list of the originally defined (Fowler, 2019) and most frequently mentioned code smells can be found in (Lacerda et al., 2020), which proposes measures to counteract them. *Data clumps* are on this list.

2.2 Data Clumps

One manifestation of code smells is *data clumps*, which are mentioned in Fowler's list (Fowler, 2019) along with countermeasures. Fowler defines them as data items that "*tend to be like children: They enjoy hanging around together*" (Fowler, 2019). These groups of data items can be found in various places such as class fields and method parameters.

For a more precise definition, (Zhang et al., 2008) and (Hall et al., 2014) may be consulted, wherein experts were asked their explanation of *data clumps*. Their resulting definition is divided into two instances: *fields* and *parameters*.

2.2.1 Fields Instance

According to (Zhang et al., 2008) and (Hall et al., 2014) a fields instance of *data clumps* is present if:

- More than three data fields are shared in two or more classes.
- The data fields have the same signatures, consisting of names, types and visibility.
- Instance fields are not necessarily found in the same order and can be distributed over an instance.

An example of *data clumps* in this instance is presented in Listing 1, which contains two classes that both share the same fields: *foo*, *bar* and *foobar*. These fields may be extracted into another class to eliminate the code smell.

Listing 1: Fields Instance Example of Data Clumps.

```
public class MyClass {
    private int foo;
    private int bar;
    private int foobar;
    public void method() { }
}
public class MyOtherClass {
    private int bar;
    private int foo;
    public void method(int c) { }
    private int foobar;
}
```

Although the definition requires that the names must be identical, in our understanding, semantic equality rather than exact equivalence is more relevant.

2.2.2 Parameters Instance

According to (Zhang et al., 2008) and (Hall et al., 2014), a parameters instance of *data clumps* is present if:

- More than three input parameters are shared in two or more method declarations.
- The input parameters have the same signatures, consisting of names, types and visibility.
- Method parameters are not necessarily found in the same order.
- The same inheritance hierarchy and method signature should not be present in these methods.

An example of *data clumps* in this instance is presented in Listing 2, which contains two classes that both employ a method in which the parameters *foo*, *bar* and *foobar* share the same signature. These parameters may be replaced with a class containing these parameters to eliminate the code smell.

Listing 2: Parameters Instance Example of Data Clumps.

```
public class MyClass {
    public void method(
        String s, int foo,
        int bar, int foobar
    ) { }
}
public class MyOtherClass {
    public void method(
        int bar, int x,
        int foo, int foobar
    ) { }
}
```

2.3 Data Clumps Refactoring

In this section, the procedure to counteract *data clumps* described by Fowler in (Fowler, 2019) is explained briefly. In the first step, Fowler suggests identifying a data clump. Thereupon, by means of *Extract Class* (Fowler, 2019), for example, the data fields are to be extracted into an object. With *Extract Class*, a class with too many fields and methods may be separated into two classes, which improves the understanding of both of them. Then, the input parameters of methods must be checked and replaced, if necessary, such as using *Introduce Parameter Object* or *Preserve Whole Object* (Fowler, 2019). *Preserve Whole Object* reduces the size of a parameter list by passing the whole object to a method instead of only the necessary parameters. *Introduce Parameter Object* shortens the size of a parameter list by grouping parameters that always appear together in method signatures into an object. Fowler argues that *Introduce Parameter Object* or *Preserve Whole Object* reveals the benefit directly apparent through the shortened parameter lists or simplified method calls.

Fowler addresses the problems and code smells resulting from the refactoring of *data clumps*. A resulting code smell from this refactoring is data class, which Fowler advises is contrary to the production of, for example, a record structure. The purpose of a data class is only to store data without further functionality. For refactoring *data clumps*, it is not problematic to use only some fields of the new objects, provided that at least two fields have been replaced with the new object. After refactoring, it is important to look for the feature envy type of code smell, which exists when the purpose of two methods in different classes is only to communicate with each other. The class newly created during the refactoring of *data clumps* may be enriched with meaningful behavior structures. This, in turn, helps to avoid many code duplications and speeds up future development (Fowler, 2019).

To the best of our knowledge, there are only two different refactoring types for *data clumps*. The first involves manual steps for *Extract Class*, *Introduce Parameter Object* and *Preserve Whole Object*. The second type involves machine learning, which is not within the scope of this study.

2.4 Abstract Syntax Tree

For the analysis of source code files, a representation of the source code in a suitable data structure is helpful. An AST is a representation of the source end in the form of a tree. Here, the individual components of the code such as expressions, operators, literals and variables are assigned to groups and appended to the root node. By means of traversing over the tree, each place in a program can be addressed in such a way.

2.5 Program Structure Interface

As a layer on top of the AST, a PSI can be used. As a layer in the IntelliJ Platform, it parses files and creates a syntactic and semantic code model. With the PSI code inside, the IDE can be highlighted. As a result, the highlighted code may present the developer with a description of the problem.

3 RELATED WORK

Several proposals have recently examined the detection and correction of code smells. Many studies and investigations (dos Santos Neto et al., 2015), (Khrishe and Alshayeb, 2016), (Mehta et al., 2018), (Palomba et al., 2018) and (Guggulothu and Abdul Moiz, 2019) have been performed to identify the optimal sequence of refactoring steps to remove a code smell. The order of detecting and refactoring code smells may have an impact on the resulting software quality. According to (dos Santos Neto et al., 2015), the literature on code smells may be categorized into four groups: *code smell detection*, *code smell correction*, *code quality evaluation* and *preservation of observable behavior*. The primary search for related work is directed at the first two groups because of the focus of our approach.

In the code smell removal experiments in (Arcelli Fontana et al., 2015), no automatic approach is suggested or named for refactoring *data clumps*. Instead, the recommended refactoring steps align with those in (Fowler, 2019). To perform these steps, the following tools are specified in (Arcelli Fontana et al., 2015): Eclipse (Eclipse Foundation, 2022), IntelliJ integrated development environment application (IDEA) (JetBrains, 2022), and RefactorIT (Aqris Software, 2016), although the last one does not support the *Extract Class* feature.

According to the review in (Lacerda et al., 2020) of eight scientific databases and the 40 identified secondary studies between 1992 and 2018, there is currently no tool for automatic refactoring for *data clumps*. To the best of our knowledge, a tool for live code smell detection with refactoring options for *data clumps* remains an innovation of this study, even from 2018 until 2022. Thus, most tools for code smells focus on detection or visualization, and only a few offer refactoring suggestions at all. According to (Lacerda et al., 2020), there are still code smells from Fowler's list for which tools with refactoring suggestions do not yet exist. Below, we present an overview of the frequently cited tools for code smell detection from (Lacerda et al., 2020), (Felix and Vinod, 2016), (Pessoa et al., 2012) and (Fernandes et al., 2016) extended by our own experiments with the following tools.

- **cASpER** (De Stefano et al., 2020) is an IntelliJ IDEA plugin that aims to assist developers in the identification and refactoring of code smells excluding *data clumps*. The plugin provides visual and semi-automated support for detecting and removing four different types of code smells.
- **CBSD** (Code Bad Smell Detector) (Hall et al., 2013), is a standalone tool that can examine Java source-code files for five types of code smells. These code smells may be found in Fowler's list (Fowler, 2019), but some have not been studied thoroughly (Lacerda et al., 2020). CBSD uses an AST approach to discover *data clumps* and other code smells. The results can be viewed in detail using an extensible markup language (XML) export or a graphical user interface (GUI).
- **CCFinder** (CCFinder, 2008) is a standalone code clone detection tool. It is token-based and uses a suffix tree matching algorithm. This tool supports various programming languages such as Java, C, C++ and others.
- **JDeodorant** (Mazinanian et al., 2016) is a plugin for Eclipse that detects 5 code smells excluding *data clumps* in Java source code and provides refactoring suggestions. This is one of the few detection tools that supports refactoring suggestions.
- **PMD** (PMD, 2023) is a tool for static analysis of Java source code. This tool scans the source code and looks for possible problems including bugs, dead code, improvable code, simplified expressions and duplicate code. This tool finds unnecessary variables, methods, statements and loops. PMD can be integrated with a variety of other tools such as Eclipse, IntelliJ IDEA and many others.
- **RefactorIT** (Aqris Software, 2016) is a plugin for Eclipse and NetBeans development environments. It offers the ability to detect and refactor code smells such as *data clumps*. RefactorIT has a limitation in comparison to other refactoring tools as it does not support the Extract Class refactoring technique, as reported in (Arcelli Fontana et al., 2015)
- **Stench Blossom** (Murphy-Hill and Black, 2010) uses visual elements to provide developers with a quick and comprehensive overview of a variety of code smells in the source code. This tool is available for Eclipse as a plugin and provides different views for visualizing eight code smells. Feedback to the developer is provided via a series of bars, in the form of petals, at the edge of the IDE editor, and the size of the petals is used for assessment and relevance to the developer. The plugin does not offer refactor suggestions. According to (Felix and Vinod, 2016) it is able to detect *data clumps*.

A detailed description of each tool is beyond the scope of this paper. These include NosePrints (Parnin et al., 2008), Borland Together (Micro Focus, 2023), inCode (Intooitus srl, 2013), inFusion (Intooitus srl, 2012) and FindBugs (Rutar et al., 2004). Some are outdated, some have low download numbers and some others cannot be found.

In the approaches examined (Stench Blossom and CBSD), the focus is only on the detection of *data clumps*. In this research, we take it a step further and demonstrate the possibility of live code *data clumps* smell detection as well as support the developers with recommendations for semi-automatic refactoring.

4 APPROACH

The live code smell detection of *data clumps* is proposed as a plugin for IntelliJ to provide support to the development of a software project by means of live, or at least fast, feedback for the developer. Our plugin, called Live Code Smell Detection (*LCSD*) is Java-based.

In the following passages, the general structure of our approach is discussed, along with the associated configuration options. This is followed by a more detailed description of our approach, which consists of three phases, illustrated in Fig. 1: **detecting**, **reporting** and **refactoring**.

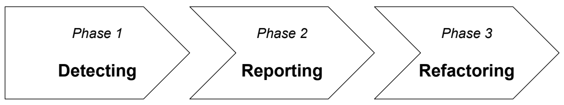


Figure 1: Phases of our approach.

In short, our tool works as follows. After an initial phase to load the plugin, the first phase, **detecting**, passes the project or files to be examined from the IDE to the tool. After the developer makes changes to the project, the changed files are examined and compared with other classes, methods and fields that are of interest for the detection of *data clumps*. There is also the possibility of extending the tool to detect other code smells.

In the second phase, **reporting**, this information is presented to the user via the IDE's interface so that they can perceive and react to the detection. In this phase, information about the positions and occurrences of possible *data clumps* is prepared and presented. If a developer decides to refactor and provides the necessary input (name for the new class), refactoring is started.

In the third phase, **refactoring**, the refactoring proposal accepted by the developer is applied. The provided name of the new class is used, and the refactoring steps suggested by Fowler (Fowler, 2019) are applied using the IDE's interfaces.

To realize these three phases our tool needs to adapt the interfaces provided by the IDE of the tool we provide the plugin for. For our approach, we first adapted these interfaces for the IDE of IntelliJ. Therefore, in the following description, we start generally from the IntelliJ application programming interface (API), which may be replaced by interfaces from other IDEs.

The simplified unified modeling language (UML) diagram of our approach with the dependency on the IDE is illustrated in Fig. 2. The class diagram has been simplified for clarity. At the core of our approach is the class *Inspection*, which is responsible for starting the process of detection. In addition, this class has the task of presenting the detected feedback to the user.

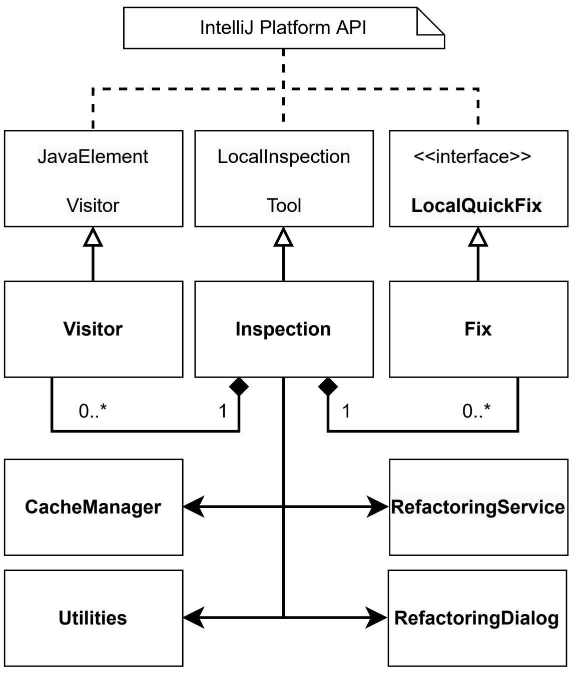


Figure 2: Class diagram of the general *LCSD* structure (notation UML 2.5).

At the initial phase to load the plugin, when opening a project, the *CacheManager* collects and prepares information about the entire project. The CacheManager allows quick access to relevant information, such as the list of PSI representations of all classes. After a change to the source code or after a scan has been performed, each *Visitor* component is presented with the affected code. A *Visitor* is responsible for scanning the AST of the source code to reveal a particular code smell. The AST is provided by the IntelliJ IDEA. A *Fix* class is responsible for performing an action to eliminate a specific code smell. As both the *Visitor* and *Fix* class are highly dependent on the class *Inspection*. Therefore both classes are illustrated as a composition. To extend our approach for a specific code smell, a *Visitor* and a *Fix* must be implemented. Consequently, multiple *Visitor* and *Fix* classes may be illustrated with the one-to-many relationship to the *Inspection* class.

After a user selects the feedback displayed by the *Inspection* component, they are guided through a refactoring dialog using the RefactoringDialog component. More details about the *data clumps* found are then presented, such as names and types of the duplicated variables with the affected methods, classes, and files. In addition, the user can enter a name for the new class. If the user approves the refactoring proposal, the related *Fix* component receives the relevant information and is responsible for manipulating the AST. There is one *Fix* component for each of the two data clump expressions, parameters and fields. The *Fix* component uses the data clump refactoring actions from the *RefactoringService* component and the generally supporting methods from the *Utilities* component, such as extracting variables, creating *getter* and *setter* methods or counting common parameters.

4.1 Detection

A correct detection of *data clumps* depends on the exact definition. For this purpose, Section 2.2 provides definitions of *data clumps*. Due to the subjectivity in detecting *data clumps*, users may have a different definition. Therefore, a configuration of parameters for the definition of *data clumps* would be useful. To support various interpretations of *data clumps*, we included a manner to configure the parameters and other settings in our approach. In our tool, the configuration can be made using the IDE interface. By default, the configuration is set according to the improved definition of *data clumps*:

- More than **three** data fields are shared in two or more classes.
- More than **three** parameters are shared in two or more methods.
- **If** the hierarchy for method parameters should be considered.
- **If** the hierarchy for data fields should be considered.
- The severity **level** of *data clumps* (i.e., whether they should be considered as information, warning or error).
- *Data clumps* are detected with a frequency of **two** repetitions.
- More than **two** groups of variables for data fields.
- More than **two** groups of variables for method parameters.

In our approach, an AST is used for the actual detection of code smells and, therefore, *data clumps*. This AST is already provided by the IntelliJ PSI at this point. For this, a *Visitor* component visits the PSI code element that represents the source code part to be examined. The *Visitor* uses methods provided by the *Utilities* component. If a method signature is examined, it is compared with all other methods in the project in terms of *data clumps*. However, if a field in a class is examined instead, the entire class and its associated fields are compared to all other classes and their fields in the project with respect to *data clumps*. Two types of scans can be distinguished: a live scan and a full scan.

4.1.1 Live Scan

Rapid feedback to the user about issues and errors may be helpful and it may be implemented via live feedback within the IDE. Information about potential errors is immediately displayed to the user, allowing this direct feedback to leverage the potential of the testing effect (Kühl et al., 2019). Live scanning support is provided using the IDE interface, and continuous scanning leads to an increased load on the system being executed. Therefore, the live scan begins only when a new source code file is opened or the user finished writing code elements that might include *data clumps* such as method signatures and class fields.

4.1.2 Full Scan

While a live scan examines only the current file and the directly associated components in a project, it might be of interest to identify all *data clumps* within the entire project. Therefore, our approach allows the option for a full analysis, which takes longer than the live scan due to the larger amount of files to examine. There are various reasons to have a project completely analyzed, such as for a performance comparison with other tools and their findings, or when an existing project should be improved. When a user completes a full scan, feedback is provided within the IDE via reporting.

4.2 Reporting

An important issue is not to disturb the developer unnecessarily, and a user should not be overwhelmed with information (Murphy-Hill and Black, 2010). Thus, the references to information, unless explicitly desired and requested, should be conveyed to the user in a subtle manner. Furthermore, a user is more familiar with managing known workflows. Therefore, in our approach, IntelliJ's workflow has been followed to report issues regarding *data clumps*. The code smells detected by our approach are presented in the identical manner in which dead code or duplicate code is identified by IntelliJ. Our goal is that the user can employ the tool to detect and remove *data clumps* in a familiar way. The affected code parts are highlighted in the editor according to the designated severity level.

Furthermore, it is possible for a user to view problems found in the project as a list within IntelliJ. This information about the detected issues or warnings is displayed in the inspection results window, which also lists other problems such as dead code or spelling errors. In this window, the occurrences of the *data clumps* are highlighted in more detail.

Figure 3 depicts an example of reporting of a class with two methods *data clumps*. These methods (*ask* and *greet*) share the three parameters of *firstname*, *lastname* and *age*. These parameters form a *data clump*, which was discovered live by our approach and is displayed to the user. Fig. 3 presents the hint in the lower area after the user has hovered their mouse over the problematic location. The user may then begin the refactoring process by selecting the Extract method.

4.3 Refactoring

After the tool has detected a code smell or data clump, the user can act on this report. The tool offers guides after the decision has been made to fix the problem. The user is informed about the issue in detail using the RefactoringDialog component and may enter a name for a new class and agree to the refactoring. The actual refactoring is based on the refactoring steps recommended by Fowler (Fowler, 2019): *Extract Class*, *Introduce Parameter Object* and *Preserve Whole Object*. First, the affected parameters or fields are extracted into a new class with private visibility. While a record class would be sufficient for this step, Fowler explicitly recommends creating a class. For this new class, a user may identify functions in other parts of the project that could be moved into the class.

- The extraction requires the new class name obtained from the user via the dialog field. For the extracted fields or parameters, new *getter* and *setter* methods are included in the created class. This provides access to the private fields.
- In the second step, all affected parameters or fields in the project are refactored to use an instance of the class created in the previous step.
- All signatures and bodies of the affected methods in parameter instances are modified to use the new class.
- In field instances, the affected fields are replaced with an instance of the extracted class.

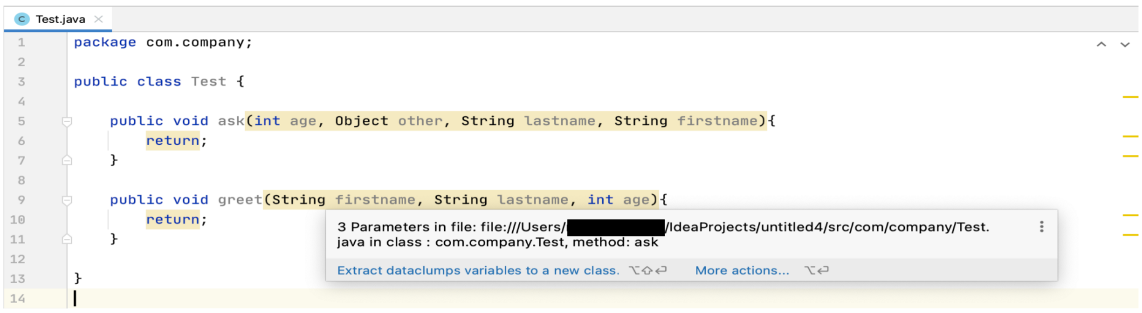
The user can revoke all modifications during the refactoring in one step, as can be done for any other action in IntelliJ IDEA.

If this refactoring is performed repeatedly, it may result in another problem: the creation of duplicate classes. To prevent this, our approach searches for suitable classes that are characterized by the fact that they have fields matching the parameter or field to be refactored. The user can then decide whether to create a new class or to use the found class. Such an option may be helpful for new developers who are involved in the project.

Fig. 4 depicts an example of refactoring of a data clump. The starting point was the code from Fig. 3. During the refactoring dialog, the user provided the information that the new file is called "Person". Fig. 4 presents two files: The file on the left corresponds to the modified original file in which the data clumps have been replaced by the newly introduced class, while the file on the right presents the automatically introduced new class with the user-defined name "Person". This class was automatically created with a constructor, private fields and associated getter and setter methods.

4.4 Extensibility

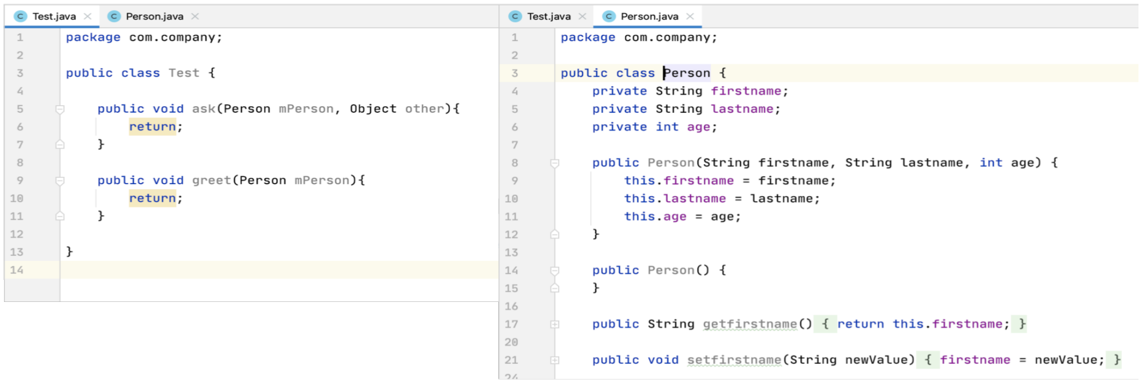
Some code smell types have common features and are highly similar, such as long method and *data clumps*. Our system architecture aims to further integrate code smell refactorings. To do so, we moved general functions to the *Utilities* component so that extensions of the tool may access them. Furthermore, extensions may use the CacheManager, in which the PSI representations of all classes in a project are maintained with the information about the super classes and interfaces. We have successfully extended our presented approach for testing purposes: The extension is for detecting and refactoring global data. We used the definition and suggestions in (Fowler, 2019) to counteract global data. The methods for generating *getter* and *setter* functions in the refactoring of *data clumps* can be reused. Further verification of the accuracy and measurement of the time required for use as a live scan has to be made.



```
package com.company;

public class Test {

    public void ask(int age, Object other, String lastname, String firstname){
        return;
    }

    public void greet(String firstname, String lastname, int age){
        return;
    }

}
```

3 Parameters in file: file:///Users/[redacted]/IdeaProjects/untitled4/src/com/company/Test.java in class : com.company.Test, method: ask

Figure 3: Example of live detected *data clumps* (before executing the refactoring step).



```
package com.company;

public class Test {

    public void ask(Person mPerson, Object other){
        return;
    }

    public void greet(Person mPerson){
        return;
    }

}
```

```
package com.company;

public class Person {
    private String firstname;
    private String lastname;
    private int age;

    public Person(String firstname, String lastname, int age) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.age = age;
    }

    public Person() {
    }

    public String getfirstname() { return this.firstname; }

    public void setfirstname(String newValue) { firstname = newValue; }
```

Figure 4: Result after executing the refactoring step for the Example of Fig. 3

5 EVALUATION

This section discusses detection accuracy. In addition to accuracy, speed is important in live code smell detection. All evaluations and tests were performed on the same computer with an Intel Core i7-6700HQ CPU and with 16 GB RAM, running a 64-bit version of Windows 10.

5.1 Accuracy

To assess the accuracy of our approach, we compared our results with those from (Hall et al., 2014). Stench Blossom does not support a full scan of entire projects with a text output, so we have not considered this tool when checking accuracy. In (Hall et al., 2014), the CBSD tool was used to search for code smells in three open-source projects, one of which is ArgoUML (version 0.26 Beta). The results for ArgoUML were then manually reviewed by two people in the study (Ferenc et al., 2020) and later were published in the Unified Bug Data Set (Ferenc et al., 2019), in which the number of occurrences of *data clumps* in the corresponding source code files was listed. For the comparison between our approach and the *data clumps* found by CBSD, the definition of *data clumps* used by CBSD has been applied, which can be summarized as follows: There must be at least three duplicates of parameters or fields on two different classes, and duplicates cannot occur between classes with hierarchical dependency. During this comparison, we noticed a challenge regarding the method for counting *data clumps*.

This challenge is illustrated in the following example: A method shares parameter duplicates in common with two other methods in different classes. Thus, the question arises, are all occurrences of the same duplicated fields and parameters counted as:

1. one *data clump*?
2. individual *data clumps*?
3. individual *data clumps*, where the original is not counted?

These different counting rules may distort the results. For the comparison, we only consider whether a data clump was detected in a file.

In the results from (Ferenc et al., 2020) for ArgoUML (version 0.26 Beta), 97 files containing *data clumps* were detected using *CBSD* after we removed non-existing file entries. Our approach found 125 files with *data clumps*. However, only 92 files were the same. We analyzed the 5 files with *data clumps* our approach did not find. For 2 files, based on manual inspection, we are confident that our approach did not miss any *data clumps*. For the remaining 3 files, the decision is not clear after manual inspection. We examined the 33 additional files in which we found *data clumps*. Among them, 2 files contain enums like classes, for which the decision is unclear whether these are data clumps. The manual examination of the remaining 31 files confirmed that the detections were *data clumps*. It is worth mentioning that there may be additional *data clumps* that are neither detected by our tool nor listed in the dataset.

Furthermore, our tests revealed several differences in the detection of data clumps between Stench Blossom, CBSD and our approach. We have created seven files for testing, each containing different *data clumps*. The seven tests were for: *data clumps* in simple parameters between two classes, *data clumps* in simple class fields between two classes, polymorphism *data clumps* where a class extends another, *data clumps* in an anonymous class, *data clumps* in interfaces, *data clumps* in inner classes and *data clumps* between two methods in a same class. In the following we will go into the tests for the tools that were not successful.

Stench Blossom could not detect the following: *data clumps* in simple class fields, *data clumps* in anonymous classes and *data clumps* in interfaces.

CBSD could not detect the following: polymorphism *data clumps*, *data clumps* in an anonymous class, *data clumps* in interfaces, *data clumps* in inner classes and *data clumps* between two methods in a same class.

We tested our tool in all those scenarios. Our tool was able to detect all of the special *data clumps* cases.

5.2 Speed

For plugins that are meant to support live code smell detection, the required time is essential. Time delays could otherwise negatively affect the workflow of a developer. This is in contrast to the analyses of existing projects in which live detection may be considered less relevant. For the measurements, we compared our approach with Stench Blossom and CBSD. Due to the fact that our approach supports both scenarios — live detection as well as a full scan — the evaluation of the speed was separated into two parts. The results of the live detection are discussed first, followed by the measurement results of full scans.

5.2.1 Live Detection

According to (Miller, 1968) and (Nielsen, 1993), reaction times up to 0.1 seconds are considered instantaneous. Response times up to 1 second are defined as the limit at which a user's thought processes are not interrupted, although a delay may be perceived. Reaction times of 10 seconds are the limit at which a user's attention persists. Therefore, for live detection of code smells, we aimed for a maximum of 1 second.

Below, we first compared our approach with Stench Blossom. The CBSD tool was ignored here, as it supports only a complete scan of a project. For the comparison with Stench Blossom, we have modified the source code so that only the detection of *data clumps* was activated, and a timer was included to measure the required time.

The open-source project ArgoUML (version 0.26 Beta) was used as the basis for the time measurement against Stench Blossom. From more than 1500 source code files, the 20 largest were used for the evaluation. We assumed that larger classes require more time than smaller ones. For this time measurement, it should be noted that the initial time of about 5 seconds to open the project in the IDE is not considered. To obtain the results, we repeated the measurements 10 times. Fig. 5 depicts the results of the time measurement as a boxplot. In this, the X-axis illustrates the *LCSD* (our approach) and Stench Blossom tools, while the Y-axis displays in seconds the time measured to open a source code file and analyze it for *data clumps*. From the figure, it can be seen that Stench Blossom produced only small deviations in the required time. In contrast, our approach revealed larger variations, reaching values 3 seconds at the upper outliers, which were caused by the largest files. However, based on the lower median of our approach, it can be concluded that less time was required for the data clump analysis for most files. For more than 50 % of all files, the time needed to scan with our approach was less than 1 second.

Furthermore, we conducted tests to assess the feasibility of live scan file analysis and evaluated the required time. To gauge its practicality, we analyzed five open-source projects of varying sizes. The smallest project, Flyway 8.1, had approximately 26 KLOC (thousands of lines of code), and the largest project, Flowable 6.7.2, contained approximately 680 KLOC. In this analysis, we selected the 20 largest files from each of the projects and measured the time needed for the analysis of data clumps. The results are depicted in Fig. 6 as a boxplot. The figure presents the five projects along the X-axis and plots the required time for analyzing a file for data clumps on the Y-axis. The median remained below 1 second in all projects. In four of five projects, for more than 50 % of all files, the required time to scan with our approach was less than 1 second. It remains open to investigate which

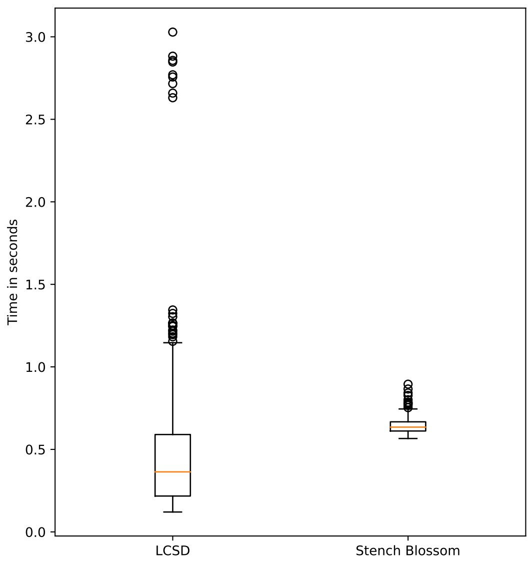


Figure 5: Time required to scan for *data clumps* per file from the 20 largest files in ArgoUML.

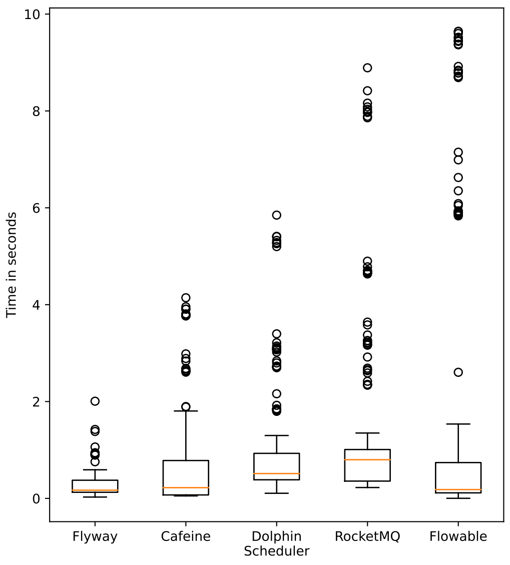


Figure 6: Time required to scan for *data clumps* per file from the 20 largest files in each project.

files were responsible for the outliers and what triggered this increased duration in the analysis.

5.2.2 Full Scan

For the investigation of the time for full scan functionality, we compared our approach with the CBSD tool. We did not consider the Stench Blossom tool here, since it does not support full scans. For the comparison, we modified CBSD so that only data clumps are analyzed, and we included a timer to measure the time required.

For the time measurement, we considered the open-source project ArgoUML (version 0.26 Beta), for which over 1500 Java files were examined completely for data clumps. To obtain the results, we repeated the measurements 10 times. For our approach, we added the initial time needed for building the cache in each case. The results are presented in Fig. 7 as a boxplot. The X-axis displays the tools LCSD (our approach) and CBSD, while the Y-axis indicates the time needed for each tool in seconds. The time required for our approach ranged from 29 to 39 seconds, with a median of 32.5 seconds, while the time needed for CBSD was between 763 and 802 seconds, with a median of 789 seconds. In all repetitions, our approach required fewer than 40 seconds to scan all files in ArgoUML for data clumps. Thus, we were able to reduce the time required by CBSD for a full scan by at least 15 times.

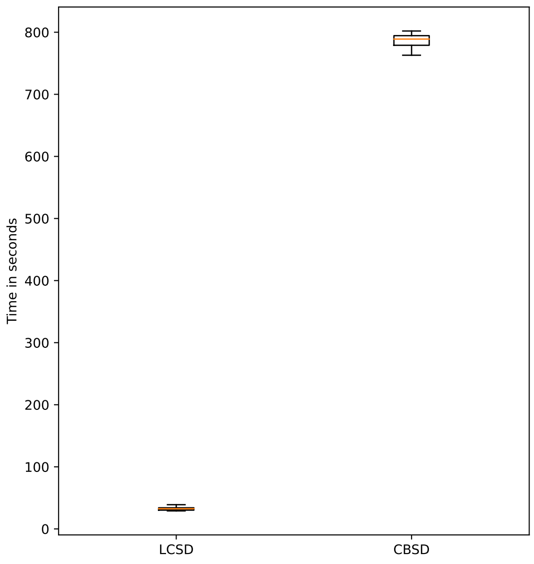


Figure 7: Time required to scan all ArgoUML project files for *data clumps*.

Furthermore, we extended the investigation of the time required for a full scan to the same five open-source projects described in Section 5.2.1. However, we did not succeed in performing the measurements with CBSD, because, for all measurements in the projects, CBSD stopped and issued errors. Thus, the following results have limited comparability with CBSD and are applicable only to our approach, the timing results for which can be seen in Fig 8 as a boxplot. The X-axis depicts the various open-source projects, while the Y-axis displays the measured time of a full scan for the respective project. The analysis reveals that the median time for the full scan of the project Flowable 6.7.2 was 706 seconds, and for the other projects, it was less than 100 seconds.

We suspect that the longer analysis time required for the Flowable 6.7.2 project, compared to the other projects, is due to the number of lines of code. Flowable 6.7.2 has approximately 680 KLOC, whereas the second-largest project, Apache RocketMQ 4.9.1, contains about 100 KLOC.

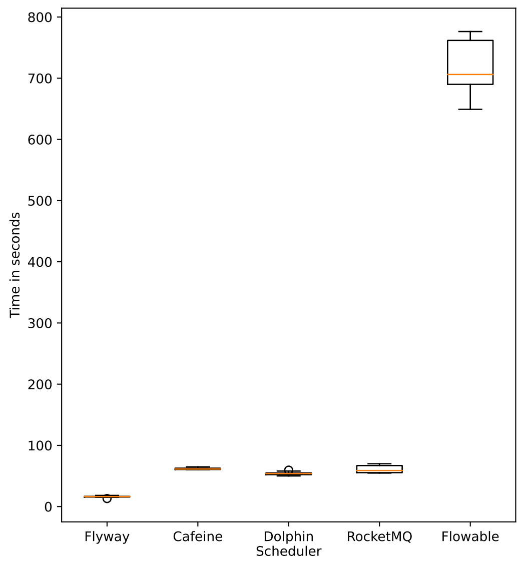


Figure 8: Time required to scan GitHub projects for *data clumps*.

6 DISCUSSION

This section discusses several limitations to the validity of our results and approach. In addition, we further address the increased usability for developers.

Two groups of users can benefit from the use of such a *data clumps* detection tool: inexperienced developers and experienced developers. The first group, inexperienced developers, may lack knowledge of best practices for writing clean code, but with the help of a *data clumps* detection tool, they can learn and improve their coding skills by identifying areas for improvement. As a disadvantage, this group of developers faces the challenge of choosing a suitable name for the new class. The second group, experienced developers, may still find value in using the tool as a quick check to confirm their code meets best practices, or to discover edge cases they may have missed. In either case, the use of a *data clumps* detection tool can help both inexperienced and experienced developers.

However, despite the perceived limitations for experienced developers, our findings highlight a crucial gap in the current research. Specifically, we found limited data on *data clumps*, with only the labeled data set from (Ferenc et al., 2020), we found hardly any other data. As a result, the significance of our results is limited with respect to other data. Accordingly, we see a need for a unified data set and with manually evaluated data clumps, where again the problem of subjective judgment arises. Similarly, in addition to the limited possibility of comparative data to measure accuracy, we found barely any data for the time needed to analyze data clumps, which is essential for the validity of live code smell detection. (Arcelli Fontana et al., 2015) reported that the varying definitions of data clumps pose an additional challenge in comparing the tool with others. One way to counteract this issue is that an appropriate tool may support parameters for the various definitions of data clumps.

The significance regarding the accuracy of our approach is limited due to the comparison with the other tools. We achieved a match of over 90 %, which raises the question of what the differences are, although the identical definition for data clumps was used as it is in CBSD. The 33 additional files detected with our approach were examined manually and found to contain 31 data clumps according to the definition. Consequently, the precise effect of different implementations for the detection of data clumps remains to be determined.

The Fig. 5 indicates that Stench Blossom could be fast enough for live detection of *data clumps* as it took less than 1 second in our measurements. The speed increase seen with our approach for live detection could be due to the use of the (maybe faster) API of IntelliJ instead of Eclipse.

In our results for timing, we note the possibility of the live code smell detection of data clumps by a tool. While in some projects and files the required response time was greater than the time defined by (Miller, 1968) and (Nielsen, 1993), at which a developer's flow of thoughts is interrupted, the median was below this threshold for all projects examined, which, to our knowledge, are not extremely complex edge cases of classes. All of our measurements were performed in an isolated test, whereas in reality, it may well be that other programs or plugins on the developer's computer may negatively affect the performance of our approach.

Furthermore, according to (Vidal et al., 2014), the order in which *data clumps* are removed is relevant. It may be helpful to first remove other code smells, which may fix the issue of *data clumps*. In this respect, our approach cannot provide any guidance on the relevance of the detected *data clumps* and the interaction of code smells.

Another challenge for the detection of *data clumps* is the identification of identical variables and parameters. In our approach, we assume that these have the same names, ignoring semantic identity beyond naming, based on the data clump definitions (c.f. 2.2). For example, a parameter xVal may have a connection to a variable named xPos, which might be identical in a semantic way. Therefore, a data flow analysis would be needed. Given that the *serialVersionUID* field has been detected in *data clumps*, it is advisable to warn the user that automatic refactoring has its limits.

Finally, despite the subtle way of presenting issues in IntelliJ's IDE, we face the question of the extent to which this type of reporting contributes to distraction. In this regard, there is a need for further research into the representation of *data clumps*. Related to this, there is still the possibility in our approach to make the developer more aware of the *data clumps* found and the potential problems involved.

7 CONCLUSION

With the approach proposed in this paper, we were able to demonstrate that live detection of data clumps is quite feasible in terms of response time. The measurements recorded median times for the analysis of data clumps below 1 second. Thus, we were able to confirm our hypothesis regarding the sufficiently fast detection of data clumps. In addition to the implementation of live detection, we successfully integrated the full scan functionality, achieving a significant increase in speed for the ArgoUML project files compared to CBSD, Stench Blossom, and other tools.

Regarding accuracy, our approach achieved a satisfactory rate of 90 % in detecting data clumps compared to CBSD. Moreover, 10 % of *data clumps* of data clumps detected through our approach were not identified by Stench Blossom or CBSD. Objective or standardized definitions of data clumps and tools may facilitate comparability of parameterization in this regard, along with clarifying how code smells are counted. Furthermore, a data set for comparing timing and accuracy, which are manually verified, would be useful. The comparison of the detection of *data clumps* using the files we created with existing *data clumps* could be extended. Additional manually created test cases could be considered from (Ferenc et al., 2020). The differences between our approach and CBSD come from the particular test cases. Since the accuracy of detection of our approach is 90 % compared to CBSD, the task of investigating where the differences in detection arise remains.

To the best of our knowledge, implementing live detection of data clumps is novel to this study. Our innovative approach supports both new and experienced developers in the creation of a project through live detection and direct refactoring suggestions.

Despite these beneficial features and significant potential in this form of support, we would like to improve our approach to semantic detection of related parameters and fields. We are planning an extension or development for the programming language JavaScript. As for future goals, we aim to provide better support for inexperienced programmers, who could increase their knowledge through examples and solid explanations. Furthermore, there is a need to consider how experienced programmers could be supported even further. We can imagine approaches aligned with continuous integration or continuous delivery with refactoring suggestions here.

REFERENCES

Aqris Software (2016). Refactorit. https://sourceforge.net/projects/refactorit/. Accessed: Feb. 04, 2023.

Arcelli Fontana, F., Mangiacavalli, M., Pochiero, D., and Zanoni, M. (2015). On Experimenting Refactoring Tools to Remove Code Smells. pages 7:1–7:8.

Becker, P., Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring - Improving the Design of Existing Code*. Addison-Wesley Professional, Boston.

Brown, W. H., Malveau, R. C., McCormick, H. W., and Mowbray, T. J. (1998). AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.

CCFinder (2008). AIST CCFinderX is a Code-Clone Detector. http://www.ccfinder.net/ccfinderxos.html. Accessed: Mar. 27, 2022.

De Stefano, M., Gambardella, M. S., Pecorelli, F., Palomba, F., and De Lucia, A. (2020). cASpER: A Plug-in for Automated Code Smell Detection and Refactoring. In *Proceedings of the International Conference on Advanced Visual Interfaces*, AVI '20, New York, NY, USA. Association for Computing Machinery.

Delchev, M. and Harun, M. F. (2015). Investigation of Code Smells in Different Software Domains. *Full-scale Software Engineering*, page 31.

dos Santos Neto, B. F., Ribeiro, M., Da Silva, V. T., Braga, C., De Lucena, C. J. P., and de Barros Costa, E. (2015). AutoRefactoring: A platform to build refactoring agents. *Expert systems with applications*, 42(3):1652–1664.

Eclipse Foundation (2022). Eclipse. https://www.eclipse.org/. Accessed: Feb. 04, 2023.

Felix, S. B. and Vinod, V. (2016). A Study on Code Smell Detection with Refactoring Tools in Object Oriented Languages. *International journal of business*, 5:38–40.

Ferenc, R., Toth, Z., Ladányi, G., Siket, I., and Gyimóthy, T. (2019). Unified Bug Dataset (1.2). http://www.inf.

u-szeged.hu/~ferenc/papers/UnifiedBugDataSet/. Accessed: Feb. 04, 2023.

Ferenc, R., Toth, Z., Ladányi, G., Siket, I., and Gyimóthy, T. (2020). A public unified bug dataset for java and its assessment regarding metrics and bug prediction. *Software Quality Journal*, 28.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., and Figueiredo, E. (2016). A Review-Based Comparative Study of Bad Smell Detection Tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, New York, NY, USA. Association for Computing Machinery.

Fowler, M. (2019). *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, Amsterdam.

Gronback, R. C. (2003). Software Remodeling: Improving Design and Implementation Quality.

Guggulothu, T. and Abdul Moiz, S. (2019). *An Approach to Suggest Code Smell Order for Refactoring*, pages 250–260.

Habra, N. and Lopez Martin, M.-A. (2006). On the use of Measurement in Software Restructuring Research. In Duchien, L., D'Hondt, M., and Mens, T., editors, *Proceedings of the International ERCIM Workshop on Software Evolution (2006)*, pages 81–89. Publication editors : Laurence Duchien, Maja D'Hondt and Tom Mens.

Hall, T., Zhang, M., Bowes, D., and Sun, Y. (2013). Code Bad Smell Detector. https://sourceforge.net/projects/cbsdetector/. Accessed: Feb. 04, 2023.

Hall, T., Zhang, M., Bowes, D., and Sun, Y. (2014). Some Code Smells Have a Significant but Small Effect on Faults. *ACM Trans. Softw. Eng. Methodol.*, 23(4).

Intooitus srl (2012). inFusion Hydrogen. https://marketplace.eclipse.org/content/infusion-hydrogen. Accessed: Feb. 04, 2023.

Intooitus srl (2013). inCode Helium. https://marketplace.eclipse.org/content/incode-helium. Accessed: Feb. 04, 2023.

JetBrains (2022). List of Java Inspections. https://www.jetbrains.com/help/idea/list-of-java-inspections.html. Accessed: Feb. 04, 2023.

Khrishe, Y. and Alshayeb, M. (2016). An empirical study on the effect of the order of applying software refactoring. In *2016 7th International Conference on Computer Science and Information Technology (CSIT)*, pages 1–4.

Kühl, S. J., Schneider, A., Kestler, H. A., Toberer, M., Kühl, M., and Fischer, M. R. (2019). Investigating the self-study phase of an inverted biochemistry classroom – collaborative dyadic learning makes the difference. *BMC Medical Education*, 19(1):64.

Lacerda, G., Petrillo, F., Pimenta, M., and Guéhéneuc, Y. G. (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 167:110610.

Mäntylä, M. V. and Lassenius, C. (2006). Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study. *Empirical Softw. Engg.*, 11(3):395–431.

Mazinanian, D., Tsantalis, N., Stein, R., and Valenta, Z. (2016). JDeodorant: Clone Refactoring. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 613–616.

Mehta, Y., Singh, P., and Sureka, A. (2018). Analyzing Code Smell Removal Sequences for Enhanced Software Maintainability. In *2018 Conference on Information and Communication Technology (CICT)*, pages 1–6.

Micro Focus (2023). Together: Visual Modeling Software. https://www.microfocus.com/en-us/products/together. Accessed: Feb. 04, 2023.

Miller, R. B. (1968). Response Time in Man-Computer Conversational Transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), page 267–277, New York, NY, USA. Association for Computing Machinery.

Murphy-Hill, E. and Black, A. P. (2010). An Interactive Ambient Visualization for Code Smells. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, page 5–14, New York, NY, USA. Association for Computing Machinery.

Nielsen, J. (1993). Chapter 5 – Usability Heuristics.

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., and De Lucia, A. (2018). A large-scale empirical study on the lifecycle of code smell co-occurrences. *Information and Software Technology*, 99:1–10.

Parnin, C., Görg, C., and Nnadi, O. (2008). A catalogue of lightweight visualizations to support code smell inspection. pages 77–86.

Pessoa, T., Brito e Abreu, F., Monteiro, M., and Bryton, S. (2012). An Eclipse Plugin to Support Code Smells Detection.

PMD (2023). PMD - An extensible cross-language static code analyzer. https://pmd.github.io/. Accessed: Feb. 04, 2023.

Rutar, N., Almazan, C. B., and Foster, J. S. (2004). A comparison of bug finding tools for Java. *15th International Symposium on Software Reliability Engineering*, pages 245–256.

Salehie, M., Li, S., and Tahvildari, L. (2006). A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws. volume 2006, pages 159–168.

Sharma, T. and Spinellis, D. (2017). A Survey on Software Smells. *Journal of Systems and Software*, 138.

Simon, F., Steinbruckner, F., and Lewerentz, C. (2001). Metrics Based Refactoring.

Vidal, S., Marcos, C., and Diaz-Pace, A. (2014). An approach to prioritize code smells for refactoring. *Automated Software Engineering*, 23.

Zhang, M., Baddoo, N., Wernick, P., and Hall, T. (2008). Improving the Precision of Fowler's Definitions of Bad Smells. pages 161 – 166.