The Lifecycle of Data Clumps: A Longitudinal Case Study in Open-Source Projects

Nils Baumgartner and Elke Pulvermüller
Research Group Software Engineering, Institute of Computer Science, Department of Mathematics and Computer Science, University of Osnabrück, Osnabrueck, Germany

Keywords: Design Smell, Code Smell Dataset, Class Diagram, Data Clumps, Code Analysis, Reporting Format.

Abstract: This study explores the characteristics of data clumps, a specific type of code smells, in software projects. Code smells are characteristics in source code which indicate a deeper problem. Data clumps are identical groups of variables in different part of the code. The lack of datasets for data clumps can make it difficult to identify and manage these sets in software projects. We developed a tool to parse source code projects into an abstract syntax tree, facilitating detailed analysis of data clumps. Our findings reveal a notable presence of data clumps forming clusters, complicating manual refactoring. In this paper, we propose a unified reporting format for *data clump* detection and provide a granular dataset for data clumps. Additionally, we outline a detection methodology that can be applied across different programming languages and frameworks. We also provide a first look into the lifecycle and evolution of data clumps, showing that data clumps either remain in projects or accumulate over time. This work provides a foundation for further research aimed at enhancing software quality through identifying and refactoring data clumps, offering a starting point for discussions and improvements in this domain.

1 INTRODUCTION

Generally, software quality degrades over time, a phenomenon known as software aging (Parnas, 1994), driven by software's increasing complexity (Lehman and Belady, 1985). Throughout the lifecycle of a software project, various challenges may arise. One of the most significant issues is the high maintenance cost of such projects (Brown et al., 1998). Additionally, compromises in software quality are frequently accepted as trade-offs between delivering an incomplete or flawed software product and minimizing the time to market (Cunningham, 1992). Beyond this, new team members who are unfamiliar with a project can inadvertently introduce design flaws, further compromising software quality.

Symptoms of bad design and implementation manifest at the code level as "*code smells*" (Fowler et al., 1999). These undesirable traits, also known as anti-patterns (Cunningham, 1992), can be removed through a process called "*refactoring.*" Refactoring involves altering the internal structure of the software without changing its external behavior. Various approaches to refactoring exist, including semi-automated or fully automated methods (Baumgartner et al., 2023; Shahidi et al., 2022; Szőke et al., 2015).

Before initiating the refactoring process, one must identify the specific code smell to be addressed. Various types of code smells have been categorized (Fowler et al., 1999), and their definitions have been refined over time (Zhang et al., 2008). Additionally, new code smell types have emerged across various domains, reflecting advancements in programming languages, methodologies, and software architectures (Delchev and Harun, 2015). As these fields evolve, some practices become obsolete, prompting the development of tools using metrics to detect such code smells (Gronback, 2003; Habra and Lopez Martin, 2006; Simon et al., 2001; Salehie et al., 2006).

Relevance. Data clumps are a specific kind of code smell involving groups of variables appearing repeatedly in different components of a software project. These groups may not always appear in the same order or have the same names even though they serve the same purpose. Consequently, these groups are difficult to identify and refactor manually. However, as (Fontana et al., 2015) discusses, the impact of code smells on software quality varies. In this vein, (Hall et al., 2014) indicated that *data clumps* affect soft-

Baumgartner, N. and Pulvermüller, E.
The Lifecycle of Data Clumps: A Longitudinal Case Study in Open-Source Projects.
DOI: 10.5220/0012313900003645

In *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2024)*, pages 15-26
ISBN: 978-989-758-682-8; ISSN: 2184-4348

ware quality, but it is unclear whether this influence is positive or negative. Additionally, this same study suggests that there may be different kinds of *data clumps*. *Data clumps* rank among the top 10 code smells, as indicated by (Lacerda et al., 2020). However, these *data clumps* are also specifically recommended for prioritization in refactoring efforts, along with five other code smells, due to their association with software faults (Zhang et al., 2011). This process underscores their significance in the realm of software quality and suggests that further investigation into the ambiguous nature of their influence could be worthwhile. There is a lack of publicly available *data clumps* datasets, as confirmed by (Liu et al., 2021) and (Zakeri-Nasrabadi et al., 2023). This gap highlights the need for more detailed datasets, as indicated by (Baumgartner et al., 2023). Such a dataset could be useful for training and validating various tools and machine learning models.

Goal. The objective of this study is to examine the software projects to better explain the *data clumps* in their lifecycle. Although it is known how to refactor *data clumps*, refactoring every instance may result in spending too much time on *data clumps* that are not harmful or that could be removed when addressing *code smells*. A better understanding of the lifecycle of *data clumps* could help in reducing unnecessary refactoring efforts and in identifying which *data clumps* should be prioritized for removal, especially those that are more problematic. This dataset can provide valuable insights for studies focused on how to prioritize the removal of *data clumps*. More specifically, the study examines the following research questions:

- **RQ1:** How should *data clumps* be counted?
- **RQ2:** What is an efficient structure for a *data clumps* dataset that helps capture the impact of *data clumps* on software quality attributes?
- **RQ3:** What characteristics do *data clumps* possess?
- **RQ4:** How do *data clumps* manifest and evolve across the lifecycle of a software project?

Hypothesis for RQ3. We hypothesize that *data clumps* tend to form larger clusters across multiple classes.
Hypothesis for RQ4. Based on the preliminary literature review and observations, we hypothesize that *data clumps* tend to increase in number over the lifecycle of a software project.

Contribution. This paper makes the following notable contributions:

- **1)** A publicly available comprehensive dataset enabling others to test and train their tools up to code line level granularity;
- **2)** A tool to detect *data clumps* given an abstract syntax tree and a visualization tool for a graph-based representation of the coupling of *data clumps*;
- **3)** A longitudinal case study reporting quantitative and qualitative evidence on when *data clumps* are introduced in software projects; and
- **4)** New characteristics of *data clumps* impacting the software quality over the lifecycle of software projects.

Organization. Initially, we provide a brief introduction to class diagrams and *data clumps* in Section 2. Subsequently, in Section 3, we give an overview of related works and state-of-the-art tools. Next, we elaborate on our approach in Section 4. In Section 5, we draw conclusions from our findings and engage in discussion, and in Section 6, we address the validity of the threads. Finally, in Section 7, we summarize the key conclusions and final impressions.

2 BACKGROUND

This section provides the background for our research and approach. To explain the lifecycle of *data clumps* in software projects, we focus on class diagrams in Section 2.1 and define *data clumps* in Section 2.2.

2.1 Class Diagrams

The design of a software system can be achieved using class diagrams, a diagram type of the Unified Modeling Language (UML), wherein classes, attributes, methods, and relationships are modeled (Object Management Group, 2007). Certain tools, such as the Visual Paradigm (Visual Paradigm, 2002), enable this modeling alongside the generation of code based on the model. A sample segment of a class diagram is illustrated in Figure 1.

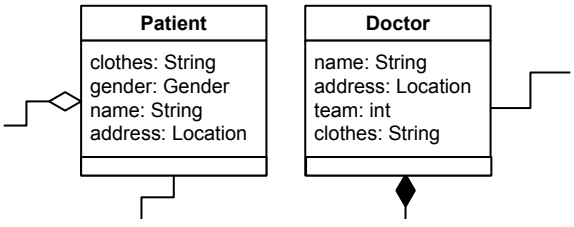

Figure 1: UML class diagram example.

An example of a part of a class diagram appears in Figure 1. In this class diagram, two classes are depicted: Patient and Doctor. The class "*Patient*" has four attributes: *clothes*, *gender*, *name*, and *address*, with no methods defined. The class "*Doctor*" involves four attributes: *name*, *address*, *team*, and *clothes*, with no methods defined, and all these fields have their data type delineated. In this illustration, a redundancy is observable in the attributes *clothes*, *name*, and *address*. This redundancy could produce a design issue manifesting as *data clumps*, a specific type of code smell.

2.2 Data Clumps

Data clumps can be categorized under the code smell taxonomy known as "*Bloaters,*" which are characterized by their propensity to inflate the source code unnecessarily. Data clumps are distinguished by groups of variables that frequently co-occur but not necessarily in a fixed sequence. In this vein, (Hall et al., 2014), hypothesized that various subtypes of *data clumps* exist.

An initial taxonomy for *data clumps* was proposed by (Zhang et al., 2008), which bifurcates these *data clumps* into field instances and parameter instances. According to this classification, parameter *data clumps* occur when at least three parameters between two methods exhibit identical structural attributes (name, type, and visibility), albeit in a permissible variable order. Conversely, field *data clumps* exist when at least three fields between two classes share identical structural attributes, again allowing for variations in sequence. Zhang et al. improved their definition based on their expert interview who states to "exclude methods inherited from parent-classes" when searching for a *data clump*. Zhang et al. added an additional criterion, when searching for *data clumps* between two methods, which states that "[t]hese methods should not [be] in a same inheritance hierarchy and [have the] same method signature." (Zhang et al., 2008, p. 164). A prototypical example of a parameter *data clump* can be schematically observed in Listing 1, where both methods share the parameters *x*, *y*, and *z*.

Listing 1: Parameters Instance Example of Data Clumps.

```
1  def calculate_distance(
2      x, y, z, speed
3  ):
4      ...
5
6  def calculate_falltime(
7      mass, x, y, z, volume
8  ):
9      ...
```

A more complex manifestation of data clumping occurs when the variables do not necessarily share identical structural attributes but are semantically congruent. For instance, in Listing 2, it can be inferred that the variables *lat*, *lon*, and *height* are semantically related to the parameters *x*, *y*, and *z* from Listing 1.

Listing 2: Parameters Instance Example of Data Clumps.

```
1  def calculate_global_distance(
2      lat, lon, height, speed
3  ):
4      ...
```

Data clumps share similarities to other code smells, such as code duplication and long parameter lists, and can easily be confused with them. While the long parameter list focuses solely on the quantity of parameters, the *data clumps* emphasize semantic similarity and recurrent appearances in disparate sections of the codebase. For a long parameter list, the analysis is confined to the local class, whereas *data clumps* may necessitate a comprehensive examination of all files within a software project.

3 RELATED WORK

This section examines related works that have closely investigated datasets for UML and *data clumps*, either through tools for detecting and refactoring or associated with software quality attributes.

In an extensive analysis by (Robles et al., 2017) of over 12 million projects, they present a comprehensive study where approximately 93,000 UML diagrams were extracted, notably featuring a single timestamp. This work underscores the prevalence and significance of UML diagrams in project documentation.

Previously, in the work of (Tahmid et al., 2016), code smell clusters were examined over time, although *data clumps* were not included. In this investigation, four main categories for code smells over time were identified. These groups were as follows: *A*: The smell existed in all examined versions, *B*: The smell did not appear initially but remained until the current version, *C*: The smell existed from the beginning and was removed later, and *D*: The smell occurred between the first and latest versions and was removed between these versions. Next, there were further subcategories of these main categories. These investigations revealed that 90 % of the smells were in categories *A* and *B*, which, according to their state-

ment, suggests that these code smells are not trivial to refactor.

An empirical study to investigate the relationship between specific code smells and design patterns was conducted by (Alfadel et al., 2020). They performed their investigation using 20 design patterns and 13 code smells, and one of these smells was *data clumps*. The analysis on 10 different Java open-source systems revealed that classes with design patterns generally displayed fewer code smells. Additionally, these researchers detected *data clumps* using *inFusion*, which ceased to exist in 2016, discovering that *data clumps* are primarily isolated from design patterns. Although those researchers focused on the co-occurence of design patterns and code smells, our work investigates the lifecycle of code smells.

In a comprehensive tertiary review, Lacerda et al. (Lacerda et al., 2020) examined the existing literature on code smells and refactoring. Their review highlighted the relationship between code smells, refactoring, and various quality attributes. However, they noted that *data clumps* have relate to the design smell of the data class, which Fowler classified as a result of the refactoring of *data clumps* (Fowler et al., 1999).

A recent study by Zakeri-Nasrabadi et al. (Zakeri-Nasrabadi et al., 2023) surveyed 45 datasets used for detecting code smells. Notably, the study revealed that six code smells, including *data clumps*, are not currently supported by any dataset. This lack of support highlights the need for datasets that include information on the impact of these smells on software quality attributes. Hall et al. (Hall et al., 2014) published a *data clump* dataset, but this link is no longer accessible, emphasizing the need for more comprehensive *data clump* datasets. Although a dataset for *data clumps* exists in the Unified Bug Dataset (Ferenc et al., 2020), *data clumps* were detected only at the file level and did not offer general information about quantity or severity, as stated by (Baumgartner et al., 2023). In this vein, (Hall et al., 2014) developed the tool *CBSD*, which can detect *data clumps*. They noted certain areas for improvement and discovered that *data clumps* were related to fewer errors in two of the three software projects they analyzed and in one case to more errors. They said their analysis was limited to the file level and did not consider the quantity or significance of *data clumps*. Additionally, they indicated that the strength of a code smell could also be important, although they did not investigate this aspect. This categorical measurement is a common standard in error prediction, where errors themselves are also categorical. Such a categorical classification of code smells, indicating whether they are present or not, loses information about the number of code smells present. Thus, there appears to be a lack of fine-grained datasets in *data clumps*, which are necessary for a more accurate assessment of whether *data clumps* are helpful or harmful.

Liu et al. (Liu et al., 2021) proposed an automatic method for selecting relevant features from source code using deep neural networks and mapping them to predictions. Additionally, they introduced a technique for generating labeled training data, essential for the effective functioning of deep neural networks. Their approach was tested on four well-known code smells, excluding *data clumps*. These authors suggested that the effectiveness of their approach could be further enhanced by making better datasets available.

One comprehensive empirical study (Tufano et al., 2015) investigated the timing and reasons behind the introduction of code smells across 200 open-source projects. Contrary to popular belief, their findings suggested that code smells primarily originate during the creation of software artifacts and not during the evolutionary phases. However, this study did not include an examination of *data clumps*.

To detect *data clumps*, several tools have been developed, including *CBSD*, *Stench Blossom*, *LCSD* (Baumgartner et al., 2023), *inCode*, and *inFusion*. The tools *inCode* and *inFusion* have been frequently cited in related work but have ceased to exist since 2016 and are not available for download anymore[1]. *CBSD* is limited to file-level analysis, as acknowledged by the authors themselves. Furthermore, CBSD supports comprehensive project scans, can be executed programmatically, and is also a standalone tool. *Stench Blossom* is an integrated development environment (IDE) plugin that provides a visualization of code smells, such as *data clumps*, at the edge of the IDE editor. The size of the visual petals indicates the relevance of the specific code smell to the developer. However, Stench Blossom does not support a full scan of an entire software project, only per-file scans. Beyond this, *LCSD* not only detects *data clumps* but also provides them semi-automatic refactoring. Furthermore, *LCSD* is an IDE plugin for IntelliJ and can be executed programmatically.

In other research, (Baumgartner et al., 2023) compared *LCSD* with *Stench Blossom* and *CBSD*. Compared with *CBSD*, *LCSD* identified more *data clumps*, which were manually inspected and confirmed. This comparison was based on *ArgoUML*, in which *CBSD* took a median time of approximately 700 seconds, whereas *LCSD* took a median time of approximately 33 seconds. When compared with Stench Blossom, *LCSD* identified *data clumps* that Stench Blossom

[1]https://www.intooitus.com/

failed to detect. Since Stench Blossom does not support full project scans, a time-based comparison was not conducted.

4 APPROACH

In this section, we first refine the definition of *data clumps*, explore the counting method necessary for the subsequent generation of the dataset. Next, we present our findings regarding the characteristics and evolution of *data clumps*.

4.1 Improving Data Clumps Definition

As previously defined by (Zhang et al., 2008), the two types of *data clumps* are class fields to class fields and method parameters to method parameters. This extension addresses the directional *data clump* from method parameters to class fields, which exist or emerge during refactoring. Hence, we suggest the following names for *data clump* types: *field-field*, *parameter-parameter*, and *parameter-field*.

Furthermore, (Zhang et al., 2008) proposed a refined definition of *data clumps* as mentioned in Section 2.2. This criterion for a *data clump* definition leads to challenges, which we will demonstrate in Listing 3.

Listing 3: Challenging Data Clump Definition.

```
1 Class A
2 Class B extends A with method foo
3 Class C extends A with method foo
4 Class D extends B with method foo
      overriding foo from B
5 Class E extends C with method foo
      overriding foo from C
```

In Listing 3 we have a family of classes where *Class A* is the parent, *Class B* and *Class C* are the children, and *Class D* and *Class E* are the grandchildren. Except *Class A*, they all have a method named *foo* which has at least three parameters. According to Zhang et al.'s definition, if we look at *Class D* and *Class E*, they would form a *data clump* with between the methods *foo* since they are not in the same inheritance, because they originate from different branches of the inheritance tree. However, this scenario contradicts the expert's suggestion from Zhang et al's interview to "exclude methods inherited from parent-classes" (Zhang et al., 2008, p. 164), as both *Class D* and *Class E* inherited the method *foo* from their parent classes (*Class B* and *Class C* respectively).

This special scenario highlights the limitations in Zhang et al.'s definition of *data clumps*, as it fails to account for inherited methods in certain situations, leading to a misrepresentation of *data clumps* in the inheritance hierarchy. We propose amending the definition to align more closely with the expert statement in (Zhang et al., 2008), which states, "exclude methods inherited from parent classes." Therefore, we aim to refine Zhang et al.'s definition to exclude methods that are overridden (but may be overloaded).

4.2 Data Clumps Counting

In addressing **RQ1:** ("How should *data clumps* be counted?"), we aim to achieve two primary objectives: (1) to establish a counting methodology for structuring a *data clump* reporting format and (2) to facilitate a robust comparison among *data clump* detection tools.

Other researchers, such as (Hall et al., 2014), have employed detectors identifying *data clumps* only at the file level. This approach was utilized in creating the Unified Bug Dataset (Ferenc et al., 2020), omitting crucial information regarding the quantity and severity of *data clumps*. This limitation was also acknowledged by (Baumgartner et al., 2023).

A more precise counting of *data clumps* necessitates a shift to the code line level, denoting the exact location of each data clump. Given the nature of *data clumps* that span classes and files, it is necessary to trace the origin and destination of each *data clump* from one class, method, and file to another.

Counting *data clumps* solely at the file level can produce incomplete comparisons between detectors. For instance, two detectors might appear to identify the same *data clumps* at the file level, yet one might detect a field *data clump* while the other identifies a parameter data clump.

Although subsets of *data clumps* are technically valid, we propose counting only the largest set to avoid redundancy and to follow the refactoring guidelines suggested by Fowler (Fowler et al., 1999). Counting subsets of *data clumps* could potentially mislead code-reviewers by indicating numerous *data clumps* between the same locations, which may be beneficial only when exploring diverse refactoring options.

Furthermore, the definition of *data clumps* given by (Zhang et al., 2008) needs refinement to encompass *data clumps* originating from method parameters to class fields (however, the reverse is not true). This amendment is crucial for accurate refactoring because introducing a parameter object could create a new *data clump* between classes, necessitating further refactoring. Thus, we define the counting of a *data clump* as follows:

Let L_1 and L_2 represent *Location 1* and *Location 2*, respectively, where both locations can be their classes or methods without a common hierarchy. However, if *Location 1* is a class, *Location 2* cannot be a method.

Let V_{L_1} and V_{L_2} denote the set of variables associated with *Location 1* and *Location 2*, respectively. If a location is a class, the set includes all fields of that class. If a location is a method, the set includes all parameters of that method.

Let $S(v_1, v_2)$ be a boolean function that compares two variables, v_1 and v_2. It returns *true* if and only if the name, type, and modifiers (as defined by (Zhang et al., 2008)) of v_1 and v_2 are identical. The function is symmetric, meaning $S(v_1, v_2)$ is equivalent to $S(v_2, v_1)$. Formally, it is defined as:

$$S(v_1, v_2) = \begin{cases} \text{true} & \text{if } name(v_1) = name(v_2) \wedge \\ & type(v_1) = type(v_2) \wedge \\ & modifiers(v_1) = modifiers(v_2) \\ \text{false} & \text{otherwise} \end{cases} \tag{1}$$

This definition in equation 1 ensures that S assesses the similarity of two variables based on their essential characteristics, contributing to the identification of data clumps in code. However, this definition overlooks cases of more complex manifestations of data clumps, particularly when variables share semantic similarities. Therefore, it should be adapted according to the user's needs and specific definitions.

For reporting purposes and the comparison previously explained, only the largest set of a *data clump* D_{L1L2} is defined as the largest set satisfying the following condition: For each $v_1 \in V_{L_1}$ and $v_2 \in V_{L_2}$, $S(v_1) = S(v_2)$. The term "largest set" refers to the set with the maximum number of variables that satisfy the given condition.

$$D_{1,2} = \max\{v_1, v_2\} \mid \begin{array}{r} v_1 \in V_{L_1}, \\ v_2 \in V_{L_2}, \\ S(v_1, v_2) \end{array} \tag{2}$$

Let $D(L)$ be the total amount of *data clumps* for a given location L, defined as follows:

$$D(L) = \sum_{i \neq j} D_{L_i L_j} \tag{3}$$

This definition of when to count a *data clump* is used in the reporting format for the dataset structure proposal in Section 4.3.

4.3 Dataset Structure Proposal

Now this study examines **RQ2:** "What is an efficient structure for a *data clumps* dataset that helps to capture the impact of *data clumps* on various software quality attributes?". A fine-grained level of detail is required to support future analyses. For other analytical focuses, one must record sufficient information.

For each report on *data clumps*, it is essential to preserve crucial information to reproduce the results and to obtain the most significant information, such as the report summary (the number of *data clumps* found and the quantity of *data clump* types). Additionally, one should retain information about the project, like the *URL*, *name*, *commit hash*, and *date*. Additionally, information about the detector utilized, a link to that detector, and the options that were employed should also be saved.

Next, this article examines the detected *data clumps* may be saved at a fine-granular level. Our proposed structure[2] employs a dictionary with a key to each detected data clump. Given the nature of *data clumps*, for each *data clump* in the dictionary, we identify the key and from which location (file, class, method) to which location (file, class, method) the *data clump* is found. Because there are different types of *data clumps*, it is necessary to track which kind of *data clumps* we might have.

For every *data clump* entry, we also store the information regarding which variables are matched to other variables, and their respective locations. This process could help other automatic refactorings match corresponding variables if, for instance, a semantic signature match is utilized instead of an exact match. For every variable, we might also save the type, name, modifiers, and position if we have this information. This data is vital for refactoring and visualizing *data clumps*. Additionally, a probability for each detected *data clump* and variable match could be useful. This probability can help to compensate for missing information, like the data types in class diagrams. Beyond this, for machine learning approaches, their certainty can be saved. This information may help with prioritizing *data clumps*.

Since we saved only the largest counted *data clump* proposed earlier, we could still use this group information to generate all subsets. This reduced saving minimizes the required space while maintaining readability. Furthermore, we can use this information to check if two *data clumps* depend on or share the same subset of variables or are independent of each other. This information may be helpful when analyzing whether one or another has greater impact on software quality attributes.

[2]https://github.com/FireboltCasters/data-clumps-type-context

4.4 Dataset Generation

To generate our dataset[3], we utilized source code data from *ArgoUML*, *Apache DolphinScheduler*, *Apache RocketMQ*, *Caffeine*, *JFlex*, *JFreeChart*, and *Xerces2 Java*. We selected these seven projects because they exhibit a sufficiently high level of maturity, as evidenced by their age, number of commits, and releases. Furthermore, these projects are well known and have been analyzed in other research. All these projects are predominantly written in Java. Table 1 summarizes the latest release; the number of Java files, classes, and interfaces; and the number of previous release versions available in the Git repository.

Additionally, we manually filtered for tags that hold only plugin support, only user manuals, or no source code at all. In *JFlex*, we omitted six tags for the *maven-jflex plugin* because it included no Java source code. Since some projects like *ArgoUML* originated from SVN, the commit date is incorrect, producing an incorrect order, so we manually adjusted the commit date by the release version number for a temporary order. At this point, we also filtered 13 tags, which included only documentation or test releases, producing real tags with source code. For the *data clumps* dataset generation, we adopted a two-step approach to accommodate other analysis tools.

Step 1: In this step, we aimed to abstract a project to be analyzed into an abstract syntax tree (AST) to support multiple types of projects like UML diagrams or Source Code Projects in different languages. For a proof of concept, our tool[4] could parse class diagrams exported in an XML file from Visual Paradigm (Visual Paradigm, 2002). For analyzing the lifecycle of *data clumps*, we parsed Java source code projects with a Git history. Our approach employed *PMD* (PMD, 2023) for parsing the source code into the AST. In our static analysis, we analyzed only the existing code.

Step 2: We utilized the algorithm of (Baumgartner et al., 2023), which had a 90 % match rate at file-level with the Unified Bug Dataset, and adapted it according to the improved *data clumps* definition in Section 4.1, to check the generated AST for *data clumps*. If dependencies are not known, they can set a probability factor based on the result of the detection. Thus, in this analysis, we set this factor to 0 because we only wanted to analyze securely detected *data clumps*. The result came in the form of our proposed dataset structure, which we defined in Section 4.3. We enhanced this algorithm by detecting *data clumps* from methods to class fields.

In total, we generated a dataset with more than 450,000 detected *data clumps*, more than 360,000 *parameter-parameter data clumps*, more than 70,000 *field-field data clumps*, and more than 15,000 *parameter-field data clumps*.

4.5 Characteristics of Data Clumps

In this section, we further investigate **RQ3:** "What characteristics do *data clumps* have?"

From the more than 450,000 examined *data clumps*, a median of three variables defined a data clump. We have developed an interactive tool for visualizing *data clumps*[5], which highlights the connections. An example of the visualization of the found *data clumps* appears in Figure 2. The files are marked in gray, the classes and interfaces in dark green, the methods in light green, and the method parameters or class fields in yellow. Upon examining the found *data clumps*, we found that there was a separated graph of the *data clumps* connections among the classes. Some classes had a *data clump* only within themselves or several classes connected by *data clumps*.

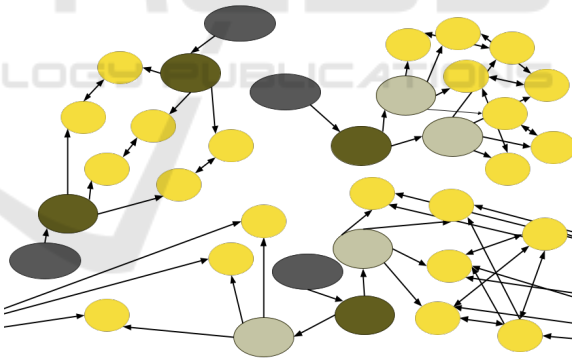

Figure 2: Excerpt of a Data Clumps Cluster Visualization.

First, at all of the examined points in time, there was more than one group of *data clumps* form a group (cluster). Next, we divided these clusters into the following three types:

- **Cluster Type 1:** A class or interface is not connected with another class or interface via any data clumps. Therefore, all the *data clumps* of that class or interface remains in the class or interface.

[3]https://github.com/NilsBaumgartner1994/Data-Clumps-Dataset

[4]https://github.com/NilsBaumgartner1994/data-clumps-doctor

[5]https://github.com/FireboltCasters/data-clumps-visualizer

Table 1: Summary of software Projects Analyzed.

Project	**Release**	**Sizes** (Class and Interface Count)	**Sizes** (Method Count)	**Maturities** (Number of git tags)	**Maturities** (development years)
ArgoUML	v0.35.1	2,259	15,564	112	25
Apache DolphinScheduler	v3.1.8	1,749	8,678	50	4
Apache RocketMQ	v5.1.4	2,002	17,011	30	6
Apache Xerces2 Java	v2.12.0	1,032	10,434	98	24
Caffeine	v3.1.8	824	6,702	68	8
JFlex	v1.9.1	700	3,075	29	20
JFreeChart	v1.5.4	1,045	10,677	7	9

- **Cluster Type 2:** Only two classes or interfaces are connected to each other via at least one data clump.
- **Cluster Type 3:** Multiple (more than 2) classes or interfaces are connected to each other via *data clumps*. They may also form chains.

In Figure 3, the percentage distribution of *data clumps* per cluster type can be taken for all 461 time points. In the median, approximately 31 % of all *data clumps* belong to *Cluster Type 1*. In contrast, in the median, about 13 % of the *data clumps* belong to *Cluster Type 2*, and around 15 % of the *data clumps* belong to *Cluster Type 3*.

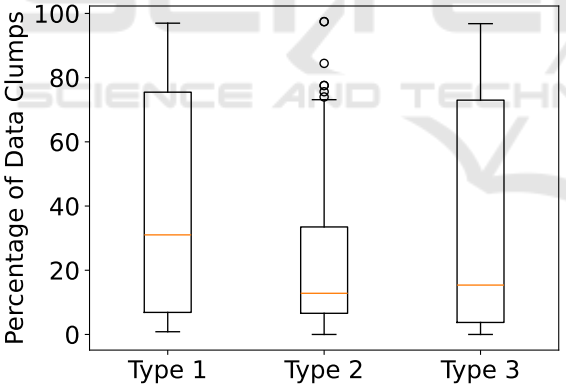

Figure 3: Distribution of Data Clump Cluster Types to the Number of Data Clumps.

Furthermore, we examined all 461 time points for Project Versions the relative number of *data clumps* types. In Figure 4, the percentage distribution of data clumps types can be taken. Additionally, the *parameter–parameter data clump* types occurred in the median to approximately 93.6 %. On the other hand, the *field–field data clump* occurred in the median at around 5.9 %. With about 0.5 %, the *parameter–field data clump* is represented in the median to the least extent.

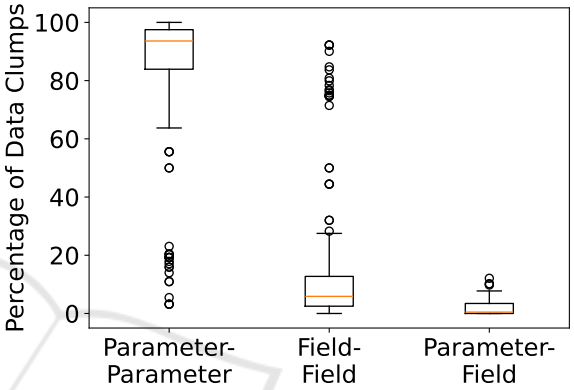

Figure 4: Distribution of Data Clumps Types.

4.6 Evolution of Data Clumps

This section explores **RQ4:** "How do *data clumps* manifest and evolve across the lifecycle of a software project?" Considering the evolution of *data clumps* in the previously introduced software projects, we examine how *data clumps* behave over time.

First, we investigate the number of *data clumps* over time. In Figure 5, the number of *data clumps* is listed for each project over the analyzed project versions. Throughout the project, the number of *data clumps* increased in the projects *Caffeine* (from 123 to 4150), *DolphinScheduler* (from 1621 to 6131), *JFlex* (from 12 to 2243), *RocketMQ* (from 986 to 4565), and *Xerces2 Java* (from 440 to 1451). The number of *data clumps* decreased in the projects *ArgoUML* (from 142 to 139) and *JFreeChart* (from 2015 to 1830).

We now further examine the categories of the evolution of *data clumps*. As (Tahmid et al., 2016) explained, the existence of code smell was divided into the following four upper categories:

- **Category A:** A code smell existed at all examined points of a project.
- **Category B:** A code smell occurred after the first examined point and remains until the current point.

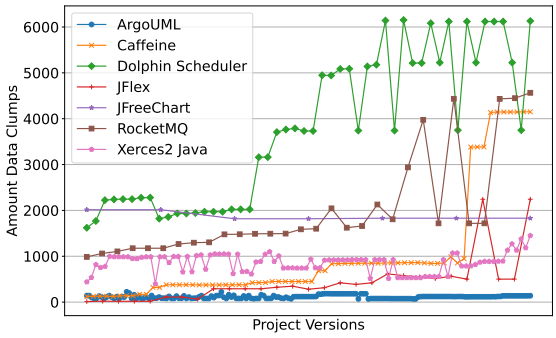

Figure 5: Number of Data Clumps across Project Versions.

- **Category C:** A code smell existed since the first examined point and was removed before the current point
- **Category D:** A code smell occurred between the first point examined and disappeared before the current point.

We examined the aforementioned projects for the four evolution categories. For this purpose, we analyzed all detected *data clumps* for each project. In Figure 6, the percentage distribution of *data clumps* to the corresponding evolution category are visible.

- Of all the *data clumps* in the project *ArgoUML*, approximately 90 % appear in *Categories C* and *D*, about 10 % in *Category B*, and none in *Category A*. Thus, many of the *data clumps* found over time may have disappeared.
- Of all the *data clumps* in the project *Caffeine*, about 80 % are assigned to *Category B*, indicating that most *data clumps* still exist. *Category C* displays about 2 % and *Category D* with about 18 % show that the remaining *data clumps* were removed over time. Finally, less than 1 % appear in *Category A*.
- In the project *Apache DolphinScheduler*, the *data clumps* are mainly distributed in *Categories B* 41 % and *D* 48 %. Approximately 11 % of all the *data clumps* are assigned to *Category C* and 0 % in *Category A*. Thus, approximately as many *data clumps* as those still present were likely removed.
- For the project *JFlex*, *data clumps* are mainly distributed in *Categories B* (57 %) and *D* (43 %). Of all *data clumps* in *JFlex* less than 1 % are assigned to *Category C* and 0 % to *Category A*. Consequently, more *data clumps* are currently present than were removed over time.
- *JFreeChart* has *data clumps* in *Categories C* (approximately 52 %) and *B* (about 48 %). Of all *data clumps* 0 % are in *Category A* and *D*. Thus, more *data clumps* were likely removed than added.
- *RocketMQ* has of all *data clumps* about 4 % in *Category A*, 59 % in *Category B*, 9 % in *Category C*, and 27 % in *Category D*.
- *Xerces2 Java* has of all *data clumps* about 0 % in *Category A*, about 27 % in *Category B*, 8 % in *Category C*, and 65 % in *Category D*. These results indicated that the majority of all *data clumps* appeared over time and then disappeared, but the number of *data clumps* overall increased.

Thus, in the projects *ArgoUML* and *JFreeChart*, the number of *data clumps* slightly decreased over time. In all other examined projects, the number of *data clumps* increased over time.

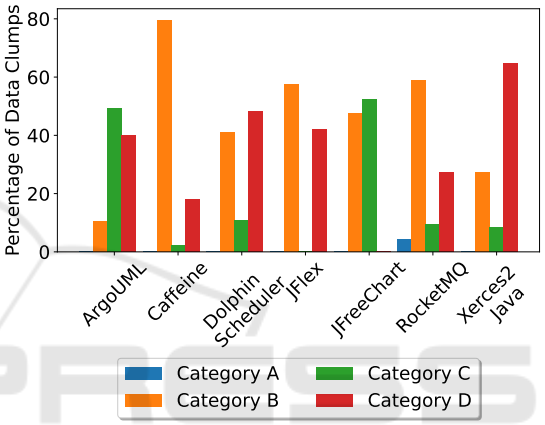

Figure 6: Data Clumps Distribution to Evolution Category.

5 DISCUSSION

Performing static analysis on class diagrams for *data clumps* allows for the early detection and mitigation of this design smell. A comprehensive examination of class diagrams from the dataset in the work of (Robles et al., 2017) remains an open endeavor, offering further points of connection.

The introduction of a probability metric for a *data clump* simplifies additional investigations concerning software quality. With an initial dataset comprising 461 time points from projects spanning 4 to 25 years, a fine-grained *data clumps* dataset has been established, which can be utilized for further research. The dataset from (Ferenc et al., 2020) can be supplemented with this data, enabling comparisons with their findings on faults.

In the examined projects, the high amount of *data clumps*, which we discovered through static analysis, indicates that prioritization of *data clumps* is necessary. An automatic incremental refactoring is desirable, yet we suspect that refactoring with too many

files simultaneously may encounter resistance and rejection. Furthermore, an investigation on other languages such as TypeScript is desirable.

Based on the analysis, in the median 31 % of our analysed *data clumps* fall into *Cluster Type 1*. Additionally, *data clumps* had a median of three common variables, which was the lower limit set based on the definition from (Zhang et al., 2008). This situation suggests that there are some *data clumps* which are simpler and easily automatable to refactor because there are no large dependencies on other classes, apart from the calls and further uses. The assumption remains that the *data clumps* of *Type 3* are harder to refactor and may potentially contribute more to code quality. That is, *data clumps* of *Cluster Type 1* might be easier to supervise and thus to maintain. The different *data clumps* cluster types facilitate further investigations into the relationship between faults in software projects. In this investigation, we did not examine these clusters in depth. Thus, the different sizes and natures of the clusters could impact software quality, as suggested by (Hall et al., 2014). It remains to be investigated to which cluster group the *data clumps* types predominantly belong, although we suspect that *parameter-parameter data clumps* predominantly belong to *Cluster Type 2*.

According to (Tahmid et al., 2016), 78.8 % of all Long Method Smell cases were (*A*+*B*). Compared with *data clumps*, we could not demonstrate these results, indicating instead that the number of *data clumps* remained approximately constant, but old ones disappeared and new ones emerged. Most of the code smells examined (once they entered the system) remained in the system, as mentioned by (Tahmid et al., 2016), corresponding to *Categories A* and *B*. In our analysis, we found that for two out of the seven projects, the majority of the data fell into *Categories A* and *B*. This was further corroborated by the increased number of *data clumps*. This circumstance raises the question of why *data clumps* are added to or linger in the examined project. These *data clumps* might positively affect quality attributes, are perhaps too difficult to refactor, or simply do not stand out due to a local removal in other folders, so they do not attract attention. This process requires further investigation, such as investigating which *data clump* types belong to the respective categories of the lifecycle. A further investigation of *Category D* could reveal whether *data clumps* reoccur after they have already been removed.

(Tahmid et al., 2016) did not account for a particular scenario that we encountered in our test cases, even though it's a rare occurrence. We suggest introducing a new category, *Category E*, to address this scenario: A code smell is present at the initial time point examined, disappears in subsequent examinations, but then reappears and persists up to the current time point.

6 THREATS TO VALIDITY

In this investigation, several potential biases and limitations were identified, which might have impacted the robustness and generalizability of our findings. Initially, our analysis was solely confined to Git tags, potentially omitting certain characteristics present during the development phase and thus altering the lifecycle analysis. Moreover, we excluded Git tags that were designated solely for documentation or did not include source code. This exclusion might be wrong if a project underwent significant changes and only saved the documentation.

Furthermore, by examining the project over time, our analysis represents merely a broad snapshot. Since we only considered Git tags, we might have missed some *data clumps* present in the commits in between, which might have been either deleted or newly added and hence not captured in this analysis. Moreover, the values derived from this work could change in the future. In addition, future research could investigate the representativeness of the projects studied, which were predominantly corporate projects rather than private or small-scale projects.

The inconsistency in the number of data points per project and the differing timelines examined also limited this study. For projects partially imported to Git like *ArgoUML*, we assumed the release order based on the semantic versioning tag, which might have been incorrect, misrepresenting the time history for this project.

Our parsing of the source codes into the abstract syntax tree (AST) did not account for dependent libraries, except for those of Java. Consequently, classes with unknown dependencies were not considered potential classes for *data clumps*. This oversight could indicate that the actual number of *data clumps* is significantly higher, affecting the results. As mentioned by (Zhang et al., 2008), additional *data clumps* might exist due to semantic similarities in names, which we did not consider due to the already high number of detected *data clumps*.

Traceability is another concern because we did not track whether a *data clump* had merely been moved or if it was new. This lack of traceability could mean it is unknown whether some *data clumps* were either merely relocated or deleted to create a new data clump.

Analyzing source code in Git repositories, while advantageous in several aspects, is limited compared with compiled class analyses. One of the primary advantages is the accessibility of source code in repositories, unlike the frequently unavailable compiled class files. This accessibility is crucial given the myriad of build tools like Maven, Ant, and Gradle that projects may employ, each with its own unique set of complexities. However, it is crucial that source code analysis does not comprehensively capture information about dependencies, which are often better explained through compiled class files containing these dependencies to other libraries. Additionally, this process is generally expedited because it bypasses the time-consuming steps of downloading dependencies and compiling the code. Despite limitations like the static nature of source code analysis and its inability to capture runtime behaviors make this approach practical for automated, large-scale examination of Git repositories.

7 CONCLUSIONS

This study investigated the characteristics of *data clumps* in software projects. By developing a tool that parses UML diagrams and source code into an AST, we facilitated a fine-grained detection and examination of *data clumps*. Our findings indicated that a significant number of *data clumps* exist in projects, and they tend to form clusters, which complicate manual refactoring. These insights emphasize the necessity for supportive and (semi-) automated refactoring of *data clumps* to avert potential design smells.

Our **RQ1** questioned the method of counting. Our manner of counting *data clumps* produces more than merely counting the files where *data clumps* occur. This process could produce the false conclusion that there are significantly more issues. Due to the directional nature of *data clumps*, both directions should be counted.

Our proposal for **RQ2** on what an efficient data structure should look like can be debated. At this point, there are points of connection that we have not considered. However, the current data structure meets the requirements set by (Hall et al., 2014) and (Baumgartner et al., 2023) for a *data clumps* dataset, which helped us examine the characteristics of *data clumps* and set a preliminary starting point.

Our hypothesis for **RQ3** was, "that *data clumps* tend to form larger clusters spanning across multiple classes," however, our findings refuted this and showed that *data clumps* are often found in individual classes. It remains open to investigate whether the cluster type has a relation to faults in the software.

Our hypothesis for **RQ4** was, "that *data clumps* tend to increase in number over the lifecycle of a software project," which, our findings confirmed in two out of the seven projects.

The ability to statistically examine class diagrams for *data clumps* provides a proactive approach to combating this design smell. This investigation suggested that a broader analysis of class diagrams from the dataset provided by (Robles et al., 2017) could provide further insights and avenues for future research. Additionally, introducing a probability metric for *data clumps* allows a deeper exploration of software quality and lays the groundwork for expanding the dataset from (Ferenc et al., 2020).

Our findings regarding the enumeration of *data clumps* and efficient data structuring offer a solid foundation for further discussions and improvements in this domain. Examining the evolution of *data clumps* in projects revealed that these clumps either persist or proliferate within projects, underscoring the need for prioritization and incremental refactoring. Exploring the relationship between *data clumps* and software quality, particularly concerning faults, remains an open field of research requiring further investigation.

In the future, we plan to continue analyzing the UML datasets from (Robles et al., 2017), specifically filtering for class diagrams and examining these diagrams' history. Analyzing the lifecycles of *data clumps*, especially concerning the four evolution categories introduced, presents a rich domain for future research. The limitations of this study, such as the exclusive analysis of Git tags and potential bias due to project selection, leave room for improvements and extended investigations.

In closing, our work provides initial insights into the nature and impacts of *data clumps* in software projects. The methods and results form the basis for subsequent research in this field, aiming to enhance software quality by identifying and remediating *data clumps*.

REFERENCES

Alfadel, M., Aljasser, K., and Alshayeb, M. (2020). Empirical study of the relationship between design patterns and code smells. *PLOS ONE*, 15:18–25.

Baumgartner, N., Adleh, F., and Pulvermüller, E. (2023). Live Code Smell Detection of Data Clumps in an Integrated Development Environment. In *International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE-Proceedings*, pages 10–12. Science and Technology Publications, Lda.

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

Cunningham, W. (1992). The WyCash portfolio management system. *ACM Sigplan Oops Messenger*, 4(2):29–30.

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

Ferenc, R., Tóth, 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(4):1447–1506.

Fontana, F. A., Ferme, V., and Zanoni, M. (2015). Towards Assessing Software Architecture Quality by Exploiting Code Smell Relations. In *2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics*, pages 1–7. IEEE.

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

Gronback, R. C. (2003). Software Remodeling: Improving Design and Implementation Quality. Whitepaper. Retrieved December 14, 2023, from https://mazure.fr/atic/tgr_softwareremodeling.pdf.

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.

Hall, T., Zhang, M., Bowes, D., and Sun, Y. (2014). Some Code Smells Have a Significant but Small Effect on Faults. *ACM Transactions on Software Engineering and Methodology*, 23(4):1–39.

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.

Lehman, M. M. and Belady, L. A. (1985). *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., USA.

Liu, H., Jin, J., Xu, Z., Zou, Y., Bu, Y., and Zhang, L. (2021). Deep Learning Based Code Smell Detection. *IEEE Transactions on Software Engineering*, 47(9):1811–1837.

Object Management Group (2007). Unified Modeling Language: Superstructure. Retrieved December 14, 2023, from https://www.omg.org/spec/UML/2.0/Superstructure/PDF.

Parnas, D. L. (1994). Software Aging. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, page 279–287, Washington, DC, USA. IEEE Computer Society Press.

PMD (2023). PMD - An extensible cross-language static code analyzer. Retrieved December 14, 2023, from https://pmd.github.io/.

Robles, G., Ho-Quang, T., Hebig, R., Chaudron, M. R., and Fernandez, M. A. (2017). An Extensive Dataset of UML Models in GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE.

Salehie, M., Li, S., and Tahvildari, L. (2006). A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 159– 168. IEEE.

Shahidi, M., Ashtiani, M., and Zakeri-Nasrabadi, M. (2022). An automated extract method refactoring approach to correct the long method code smell. *Journal of Systems and Software*, 187:111221.

Simon, F., Steinbruckner, F., and Lewerentz, C. (2001). Metrics Based Refactoring. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pages 30–38.

Szőke, G., Nagy, C., Fülöp, L. J., Ferenc, R., and Gyimóthy, T. (2015). FaultBuster: An Automatic Code Smell Refactoring Toolset. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE.

Tahmid, A., Nahar, N., and Sakib, K. (2016). Understanding the Evolution of Code Smells by Observing Code Smell Clusters. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. (2015). When and Why Your Code Starts to Smell Bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*.

Visual Paradigm (2002). Visual Paradigm. Retrieved December 14, 2023, from https://www.visual-paradigm.com/.

Zakeri-Nasrabadi, M., Parsa, S., Esmaili, E., and Palomba, F. (2023). A Systematic Literature Review on the Code Smells Datasets and Validation Mechanisms. *ACM Computing Surveys*, 55(13s):1–48.

Zhang, M., Baddoo, N., Wernick, P., and Hall, T. (2008). Improving the Precision of Fowler's Definitions of Bad Smells. In *2008 32nd Annual IEEE Software Engineering Workshop*, pages 161 – 166. IEEE.

Zhang, M., Baddoo, N., Wernick, P., and Hall, T. (2011). Prioritising Refactoring Using Code Bad Smells. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 458–464. IEEE.