




Considerations in Prioritizing for Efficiently Refactoring the Data Clumps Model Smell: A Preliminary Study

Nils Baumgartner¹ ^a, Padma Iyengar^{2,3} ^b and Elke Pulvermüller¹ ^c

¹Software Engineering Research Group, School of Mathematics/Computer Science/Physics, Osnabrück University, 49090 Osnabrück, Germany

²innotec GmbH, Hornbergstrasse 45, 70794 Filderstadt, Germany

³Faculty of Engineering and Computer Science, HS Osnabrueck, 49009 Osnabrück, Germany

Keywords: Data Clumps, Model Smell, Refactoring, Prioritizing, Systematic Approach, Weighted Attribute, Threshold-Based Priority.

Abstract: This paper delves into the importance of addressing the data clumps model smell, emphasizing the need for prioritizing them before refactoring. Qualitative and quantitative criteria for identifying data clumps are outlined, accompanied by a systematic, simple but effective approach involving a weighted attribute system with threshold-based priority assignment. The paper concludes with an experimental evaluation of the proposed method, offering insights into critical areas for developers and contributing to improved code maintenance practices and overall quality. The approach presented provides a practical guide for enhancing software system quality and sustainability.

1 INTRODUCTION

Code smell refers to specific structures in the source code that may indicate a deeper problem and compromise the maintainability and readability of software. There are various types of code smells, each pointing to potential issues in the design or implementation of software. Examples of code smells include, *data clumps*, *god class*, *duplicated code*, *large class* and *feature envy*. For instance, data clumps are a code smell where groups of data fields frequently appear together, signalling potential redundancy and suggesting the need for encapsulation or abstraction to improve code maintainability and flexibility (Fowler, 1999).

Model smells extend the concept of code smells to the architectural level, encompassing issues that affect the overall structure and design of the software model. *Data clump model smell* refers to a recurring pattern where multiple data fields consistently co-occur across various entities within a software model, indicating a potential design issue that can be addressed through refactoring for enhanced clarity and main-

tainability. *Refactoring* (Fowler, 1999) is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour.

Data clump model smell in UML (Unified Modeling Language) class diagrams occurs when multiple classes or methods share a set of attributes in fields or parameters, indicating a potential design flaw. For example, if several classes (e.g., *Person*, *Business*, *ContactInfo*) share a set of the same attributes (*street*, *city*, *postalCode*), as depicted in Figure 1. This suggests a data clump model smell, urging consideration for refactoring. In this example, a possible refactoring might be to create a class *Address* with the shared set of the same attributes.

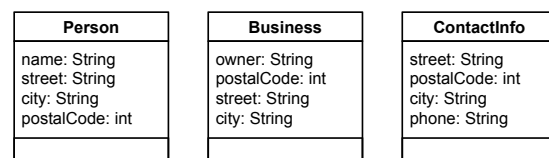





Figure 1: Example of classes sharing the same set of attributes.

While traditional code smells focus on improving individual code snippets, model smells address larger-scale design concerns that impact the software's ar-

^a  <https://orcid.org/0000-0002-0474-8214>

^b  <https://orcid.org/0000-0002-1765-3695>

^c  <https://orcid.org/0009-0000-8225-7261>

chitecture. Efficiently refactoring the data clumps model smell is crucial for improving code maintainability and readability. This practice helps eliminate redundancy, enhances code structure, and promotes a more modular and scalable design, leading to a more maintainable and adaptable software system. However, not all instances of these data clumps model smell are created equal, necessitating a systematic approach to prioritize and address the most critical issues first, for instance by refactoring. Thus, efficiently refactoring the data clumps model smell (e.g. by prioritizing) is crucial for enhancing code quality, simplifying maintenance, and promoting scalability.

In the aforesaid context, this paper explores the considerations in prioritizing for efficiently refactoring the data clumps model smell and provides the following novel contributions.

- The importance and benefits of addressing the data clumps model smell is outlined. The need for prioritizing data clumps refactoring is discussed.
- Qualitative and quantitative criteria for identifying data clumps are elaborated. The metrics to measure the quantitative criteria are described with examples.
- A systematic, customizable, simple but effective method of a weighted attribute system with threshold-based priority assignment for systematically prioritizing data clumps model smells is discussed.
- An experimental evaluation of the proposed method for the quantitative criteria is presented.

In summary, the approach presented in this paper offers a systematic and customizable method for prioritizing data clumps model smell, providing developers with valuable insights into critical areas that require attention. By combining attribute weighting, threshold-based priority assignment and sorting, our approach contributes to improved code maintenance practices and overall code quality. The flexibility of the system allows for seamless integration into diverse software development environments. Further, the proposed considerations aim to provide a practical guide for software practitioners seeking to enhance the overall quality and sustainability of their software systems.

The remainder of the paper is organized as follows. Next to this introduction section, related work is presented in section 2 and explaining the need for prioritizing data clumps refactoring. The qualitative and quantitative factors for identifying data clumps are outlined in section 3. Experimental results are discussed in section 4. Conclusion and insights for future work are presented in section 5.

2 RELATED WORK AND INFERENCES

In this section, related work on model smells in general, data clumps model smells in model representations (e.g. UML diagrams) and prioritization approaches for code/model smells are discussed. Based on a survey of the related work in the literature, some key insights on benefits of addressing data clumps model smell and the need for prioritizing data clumps refactoring are also outlined briefly.

2.1 Model Smell

The idea of model smell was elaborately discussed in (Eessaar and Käosaar, 2019). In this paper, a model smell is defined as an indication of potential technical debt in system development, hindering understanding and maintenance; this paper presents a catalogue of 46 model smells, highlighting their general applicability beyond code smells, with examples grounded in system analysis models.

Model smells appear in various model representations, such as UML¹, Simulink², and LabVIEW³, highlighting their prevalence across popular modelling platforms. In the literature, several approaches are proposed for model smell detection, underlining the ongoing efforts to address these issues in diverse modelling contexts. For instance, in (Doan and Gogolla, 2019) an enhanced version of a custom-defined tool incorporating reflective queries, metric measurement, smell detection and quality assessment features for UML representations is presented. In this work, design smells are stored as XML files, each entry containing elements like name, description, type, severity, definition, and context. However, an experimental evaluation is not provided in this paper. In (Popoola and Gray, 2021), an analysis of smell evolution and maintenance tasks in Simulink models reveals that larger models show more smell types, increased smell instances correlate with model size, and bad smells are primarily introduced during initial construction. It was inferred that adaptive maintenance tasks tend to increase smells, while corrective maintenance tasks often reduce smells in Simulink models. Similarly, in (Zhao et al., 2021), a survey-based empirical evaluation of bad model smells in LabVIEW system models is presented. The study explores model smells specific to LabVIEW systems models, revealing diverse perceptions influenced by

¹<https://www.uml.org/>

²<https://www.mathworks.com/help/simulink/>

³<https://www.ni.com/documentation/en/labview/>

users' depth of knowledge, providing valuable recommendations for practitioners to enhance software quality.

2.2 Data Clumps

Martin Fowler initially provided broad definitions for various code smells, which are generally applicable but not sufficiently detailed for automated analysis and refactoring (Fowler, 1999). In the study (Zhang et al., 2008) the definitions of selected code smells, including data clumps, were examined and refined. This research included conducting expert interviews to achieve a uniform consensus on these definitions.

Building on the improved definition for data clumps, (Baumgartner et al., 2023) introduced a plugin for the first time that enables live detection of data clumps with semi-automatic refactoring capabilities. The research demonstrated that, for different open-source projects, the time required for analysis remained under one second on average. However, the selection of data clumps to be refactored still requires manual initiation. This development represents a significant step forward in the practical application of these refined definitions in real-world software development scenarios.

In the field of software development, projects evolve over time, leading to changes in software quality, both positive and negative. These changes in projects over time result in various life cycles of code smells and model smells. The work of (Baumgartner and Pulvermüller, 2024) focuses on the analysis and examination of data clumps throughout their temporal progression. Their findings reveal that data clumps tend to group together into what are known as clusters. These clusters are characterized by multiple classes being interconnected through data clump code smells. One of the challenges highlighted by the authors is the challenge in refactoring these connections, as it requires making decisions on how to resolve each of these links. In their study, they analyzed seven well-known open-source projects, considering up to 25 years of their development history. The results indicate that, over time, the number of data clumps tends to increase in almost all the projects examined. This observation underscores the ongoing challenge in managing and improving software quality in evolving software projects.

2.3 Prioritization of Model Smell

While, the refactoring process (Fowler, 1999) enhances the software design by modifying the structure of design parts impaired with model smells with-

out altering the overall software behaviour, handling these smells without proper prioritization will not produce the anticipated effects (AbuHassan et al., 2022). Several approaches exist in the literature in the aforesaid direction of prioritization, of which some are discussed below.

In (Zhang et al., 2011) the need for prioritization of code smells is outlined. An approach based on developer-driven code smell prioritization is presented in (Pecorelli et al., 2020). In this paper, the authors perform a first step toward the concept of developer-driven code smell prioritization and propose an approach based on machine learning to rank code smells according to the perceived criticality that developers assign to them. The solution presented has an F-Measure up to 85% and outperforms the baseline approach. In (AbuHassan et al., 2022) prioritization of model smell refactoring in UML class diagrams using a multi-objective optimization (MOO) algorithm is discussed. While the authors claim that the work presented achieves longer refactoring sequences without added computational cost, it does not specifically concentrate on addressing the data clumps model smell. (Alkharabsheh et al., 2022) introduces a multi-criteria merge strategy for prioritizing the design smell of god classes in software projects, employing an empirical adjustment with a dataset of 24 open-source projects. The empirical evaluation highlights the need for improvement in the strategy, emphasizing the importance of analysing differences between projects where the strategy correlates with developers' opinions and those where there is no correlation. Prioritization of model smell refactoring using a covariance matrix-based adaptive evolution algorithm is discussed in (AbuHassan et al., 2022), where the proposed solution leads to longer refactoring sequences at no additional computational cost. However, an approach for prioritization for data clumps model smell is not available in the literature.

2.4 Inferences

In this subsection, we examine the importance of addressing data clumps model smells in software development, which are groups of frequently used data items in code. We highlight the benefits of prioritizing the refactoring of data clumps

2.4.1 Addressing Data Clumps Model Smell

From our review of existing literature and related work, we derive insights on the significance and advantages of addressing data clumps model smells. They are listed below:

- Enhanced code maintainability through the consolidation of related data, making it easier to understand and maintain.
- Improved modularity and flexibility by organizing related data into separate structures, promoting a more adaptable design.
- Reduced code duplication by centralizing common data structures, minimizing redundancy related to data clumps.
- Adherence to design principles, such as the Single Responsibility Principle, by separating concerns related to data representation.
- Efficient resource utilization through streamlined data structures, optimizing resource allocation for data associated with the data clumps smell.

2.4.2 Prioritizing Data Clumps Before Refactoring

The need for prioritizing and adopting a systematic approach in addressing the data clumps model smell stems from several reasons:

- **Efficient Resource Utilization.** By prioritizing, development teams can allocate resources effectively, addressing the most critical instances first to maximize impact and minimize technical debt.
- **Systematic Handling of Issues.** A systematic approach allows for a structured and organized way of identifying and addressing data clumps, preventing ad-hoc or inconsistent fixes and ensuring a comprehensive solution.
- **Scale and Complexity.** In large codebases, there might be numerous occurrences of data clumps. Prioritization helps manage the scale and complexity by tackling the most impactful instances initially.
- **Risk Mitigation.** Identifying and addressing critical data clumps early reduces the risk of future maintenance challenges, enhancing code quality and reducing the likelihood of introducing new issues.

Possible approaches to prioritizing and systematically addressing data clumps include:

- **Weighted Scoring.** In this approach, weights are assigned to different factors such as impact on maintainability, code duplication, and violation of design principles (to name a few) to prioritize instances with higher scores.
- **Business Impact Analysis.** Using this approach, the impact of data clumps on critical business functions can be analysed. Instances that have a

higher impact on strategic objectives can be prioritized.

- **Collaborative Decision Making.** By this approach, one can involve developers, architects, and other stakeholders in the prioritization process. Collective insights can contribute to a more comprehensive and informed decision-making process.
- **Historical Records and Use of Artificial Intelligence (AI).** The historical records of code maintenance can be analysed to identify instances causing frequent issues or requiring frequent modifications, prioritizing these for refactoring. When such a metric dataset is available for large code bases, then an AI/Machine Learning (ML) approach can be used to integrated to enhance the prioritization of data clumps model smells.

In summary, the analysis of existing literature and related work provides valuable insights into the importance of addressing data clumps model smells. The benefits include enhanced code maintainability, improved modularity, reduced code duplication, adherence to design principles, and efficient resource utilization. To effectively address data clumps, prioritization and a systematic approach are crucial. Prioritization ensures efficient resource allocation, systematic issue handling, scalability management in large codebases, and risk mitigation by addressing critical data clumps early on, enhancing overall code quality.

Notably, there is a lack of a systematic, customizable, and effective method for prioritizing data clumps model smell. The work discussed in this paper addresses this gap, introducing a weighted attribute system with threshold-based priority assignment, providing a comprehensive solution to systematically prioritize data clumps before refactoring.

3 CRITERIA FOR IDENTIFYING DATA CLUMPS MODEL SMELL

Identifying data clumps model smell involves evaluating both qualitative and quantitative factors to ensure a comprehensive assessment of the software's quality and refactoring needs. Qualitative factors offer insights into subjective aspects. On the other hand, quantitative factors provide measurable data for a more precise evaluation. Both these factors are discussed below. Further, the quantitative factors are discussed in detail, accompanied by examples and specific metrics for each criterion, contributing to a systematic approach to identifying and addressing data clumps.

3.1 Qualitative Factors

Qualitative factors typically involve characteristics that are descriptive, subjective, and not easily quantifiable in numerical terms. In the context of software development, qualitative factors often capture aspects related to strategic alignment, maintainability impact, adaptability to changes, and feedback from the development team. These factors provide valuable insights into the overall quality, alignment with goals, and collaborative aspects of the software, which may not be expressed solely through quantitative metrics but involve subjective evaluations and considerations.

- **Business-Critical Functions.** Prioritizing data clumps within classes related to essential business logic or critical functionalities aligns with strategic goals.
- **Security and Compliance.** Prioritizing data clumps within classes related to crucial security aspects, contributing to support a more reliable software.
- **Impact on Maintainability.** This criterion involves detecting data clumps with a substantial influence on the maintainability of the codebase, contributing to overall codebase health and facilitating future modifications.
- **Integration with Other Systems.** Examining how well the software integrates with existing systems and third-party services, which can affect its functionality and the efficiency of workflows.
- **Technical Debt Management.** Prioritizing the refactoring of classes with high technical debt is crucial for future development efforts. This includes understanding the potential costs and risks associated with delaying necessary updates or refactorings.
- **Strategic alignment and Architecture Vision.** Ensuring that refactoring for data clumps aligns with the overall architectural vision promotes consistency and adherence to design principles.
- **Adaptability to Changes.** Prioritizing refactoring for data clumps hindering the system's adaptability to evolving requirements ensures ease of accommodation for changes.
- **Feedback from Development Team.** Incorporating team feedback and prioritizing data clumps identified as challenging or hindering ensures that improvements address real pain points and enhance developer efficiency.

3.2 Quantitative Factors

Quantitative factors refer to measurable and numerical characteristics that can be assigned specific values or quantities. In the context of software development and refactoring, quantitative factors often involve metrics or measurements that provide objective data. These factors can be quantified, allowing for a more precise and numerical evaluation of various aspects of the codebase.

In the given context, factors like the widespread occurrence, complexity, dependencies, consistency with design patterns, and degree of code duplication involve measurable aspects for each data clump that could contribute to the overall assessment of the code quality and refactoring needs.

Normalization Score

For experimental evaluation, a normalized metric for each of these criteria on a scale of 0 to 10 for the mentioned factors, we follow a consistent normalization approach for each factor. So for each of the five quantitative aspect, the formula below is used to obtain a normalized and consistent score:

$$NS := \left(\frac{\text{Actual Score}}{\text{Max. Possible Score}} \right) \times 10 \quad (1)$$

The components in the formula in (1) are described below:

- **Normalized Score (NS).** This is the final score that is derived from the actual score and the maximum possible score. It represents a scaled value on a scale of 0 to 10, providing a standardized measure for comparison.
- **Actual Score.** This is the real or observed value for the specific metric being evaluated. It could be the number of attributes, occurrences, or any other measurable quantity related to the data clump model smell.
- **Maximum Possible Score.** This represents the highest or most favourable value that the metric could achieve. It acts as a reference point for scaling the actual score. For example, if the metric is the number of attributes or parameters, the maximum possible score might be determined by the total number of attributes or parameters a class can ideally have.
- **Scaling Factor (10).** The multiplication by 10 is used to scale the normalized score to a range of 0 to 10. This standardizes the scores across different metrics, making them easier to compare.

Thus, the formula in (1) calculates the normalized score by dividing the actual score by the maximum

possible score, and then scaling the result to a range of 0 to 10. This normalization process helps in creating a consistent and comparable assessment across various metrics used to evaluate data clumps.

3.2.1 Widespread Occurrence

This criterion counts the frequency of occurrence of data clumps that appear across multiple classes, ensuring a comprehensive impact on code quality and system consistency.

- **Metric:** Count the number of classes in which the data clump appears.
- **Example:** Let's consider an example, as depicted in Figure 2, where the data clump of class A is widespread across 5 classes out of a maximum possible count of 11 classes. The connection lines in this example are data clumps. Then the normalized score is:

$$NS = \left(\frac{5}{11}\right) \times 10 = 4.54 \quad (2)$$

So, in this case, the data clump's *widespread occurrence* factor has a normalized score of 4.54 on a scale of 0 to 10. This indicates that the data clump is present in a significant portion of the classes but not in all of them.

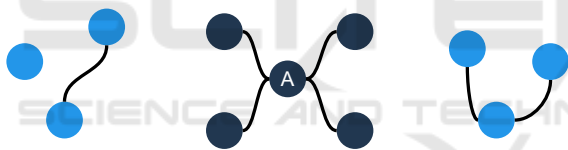


Figure 2: Widespread occurrence of data clumps.

3.2.2 Size of Attributes or Parameters

Addressing large and intricate data clumps within classes early on is crucial for achieving more significant improvements and simplifications in the system.

- **Metric.** Measure the number of attributes or parameters within the data clump relative to the total number of attributes or parameters.
- **Example.** A class A as depicted in Figure 3 has 4 attributes. The data clump shows 3 shared attributes. The normalized score for a scenario where a data clump has 3 attributes out of 4, is calculated as follows:

$$NS = \left(\frac{3}{4}\right) \times 10 = 7.5 \quad (3)$$

The normalized score of 7 signifies a high level of complexity and size associated with this data clump. Such complexity could impact code readability, maintainability, and overall system robustness.

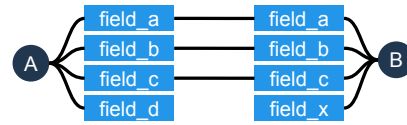


Figure 3: Size of attribute occurrences of data clumps.

3.2.3 Dependencies and Coupling

Tackling data clumps that contribute to tight coupling and complex dependencies early in the process enhances modularity and mitigates the risk of unintended consequences.

- **Metric.** Analyse the number of dependencies or associations between the data clump and other classes.
- **Example.** Assume there are 40 classes in total. The data clump is found in dependencies across 25 classes. Then, the normalized score is:

$$NS = \left(\frac{25}{40}\right) \times 10 = 6.25 \quad (4)$$

In this case, the data clump's "dependencies and coupling" factor has a normalized score of 6.25 on a scale of 0 to 10. This suggests that the data clump is moderately coupled with a substantial number of classes, indicating some level of tight coupling and complex dependencies.

3.2.4 Consistency with Design Patterns

Prioritizing refactoring efforts that do not align with established design patterns or best practices ensures a standardized and well-structured approach to resolving data clumps.

- **Metric.** Evaluate how bad the data clump adheres to established design patterns. Use a subjective assessment or a set of criteria to assign a score. For example, for a data clump, we may count how many of our tracked design patterns are followed.
- **Example.** For example, out of the maximum 10 design patterns considered, if the data clump aligns with 3 of them only, then the data clump gets a high normalized score of 7 as determined below:

$$NS = \left(\frac{10 - 3 = 7}{10}\right) \times 10 = 7 \quad (5)$$

3.2.5 Degree of Code Duplication

This is a measure of how much the data clump contributes to code duplication across classes.

- **Metric.** Count the number of classes in which the data clump leads to a significant code duplication

- **Example.** Suppose the data clump contributes to code duplication in 20 different classes out of a maximum possible count of 30 classes. Then the normalized score is 6.67 as calculated below:

$$NS = \left(\frac{20}{30}\right) \times 10 = 6.67 \quad (6)$$

In this example, the data clump's *degree of code duplication* factor has a normalized score of approximately 6.67 on a scale of 0 to 10. This suggests a substantial, but not overwhelming, degree of code duplication caused by the data clump across classes. This metric is different from size of attributes or parameters since, the degree of code duplication considers the code within the classes.

4 EXPERIMENTAL EVALUATION

In this section, an experimental evaluation of the prioritization approach proposed in this paper is discussed in detail. This approach takes as input the data clumps metrics tuple, which is defined in section 4.1 for each. The algorithm used for the prioritization of data clumps is described in section 4.3.

4.1 Data Clumps Metrics Tuple

These metrics for data clumps are corresponding to the respective attributes for the quantitative factors mentioned in section 3.

Thus, each data clump metric is a defined as a tuple **Data Clump Metrics Tuple** (δ): (Name, WO, SZ, DP, CDP, DC), where

- **Name:** A unique identifier or label for the data clump.
- **WO:** Widespread Occurrence - Indicates the frequency of occurrence of the data clump across multiple classes.
- **SZ:** Size - Represents the number of attributes or parameters within the data clump.
- **DP:** Dependency - Reflects the level of dependency of the data clump on other components.
- **CDP:** Consistency with Design Patterns - Measures the consistency of the data clump with design patterns.
- **DC:** Degree of Code Duplication - Indicates the extent of code duplication within the data clump.

Let us consider an example instance of the data clump metric tuple, $\delta_1 = (\text{"DataClump1"}, 8, 3,$

$5, 4.5, 9)$. The following provides brief explanation for each metric score in this data clump tuple δ_1 with the name *DataClump1*.

- **WO** - 8: A score of 8 indicates that this data clump is frequently present across multiple classes. It suggests that the data clump has a significant impact on code quality and system consistency due to its widespread use.
- **SZ** - 3: The score of 3 implies that the data clump has a moderate number of attributes or parameters. While not excessively large, it still contributes to the size of the data clump, impacting maintainability and readability.
- **DP** - 5: With a score of 5, this data clump exhibits a moderate level of dependency on other components. This suggests that changes to the data clump may have implications for other parts of the system, influencing overall system complexity.
- **CDP** - 4.5: The score of 4.5 indicates a reasonably good consistency of the data clump with design patterns. It suggests that the structure of the data clump aligns fairly well with the established design principles.
- **DC** - 9: A score of 9 reflects a high degree of code duplication within the data clump. This implies that there is a significant amount of redundant code, which can negatively impact maintainability and increase the risk of errors.

4.2 Weights and Thresholds

In the proposed approach, weights and thresholds are pivotal elements in the prioritization of data clumps, providing a mechanism to customize and refine the refactoring process. These parameters influence the assignment of priorities to individual data clumps based on their quantitative factors.

4.2.1 Weights

Weights are assigned to qualitative factors associated with data clumps, reflecting their relative importance in the prioritization process. Each factor, such as widespread occurrence, size, dependency, consistency with design patterns, and degree of code duplication, is assigned a weight. Higher weights signify a greater influence on the overall prioritization. It is to be noted that the sum of weights is less than or equal to 1.

In our approach, weights are defined in the weights data structure as shown below. This allows developers to tailor the prioritization based on project-specific considerations. For example:

```
weights = {
    'widespread_occurrence': 0.25,
    'size': 0.3,
    'dependency': 0.2,
    'consistency_DP': 0.1,
    'degree_codeDuplication': 0.15
}
```

These weights are used in the calculation of the weighted score for each data clump, providing a customizable approach to emphasize specific factors.

4.2.2 Thresholds

Thresholds are predefined values that categorize data clumps into distinct priority levels, such as "High Priority," "Medium Priority," and "Low Priority." These thresholds enable a quantitative classification based on the calculated weighted scores, guiding developers in identifying critical refactoring candidates.

In the provided algorithm, thresholds are defined in the `thresholds` data structure as follows.

```
thresholds = {
    'high': 8,
    'medium': 6,
    'low': 4
}
```

Adjusting these thresholds allows developers to set criteria for high-priority refactoring based on the project's specific requirements and objectives. The flexibility of weights and thresholds enhances the adaptability of the prioritization process across different software development scenarios.

4.2.3 Weighted Score

The weighted score is calculated for each instance of the *data clump metrics tuple* based on the specified weights for different quantitative factors. The formula for calculating the weighted score is as follows:

$$\text{Weighted Score} := \sum_i (\text{weight}_i \times \text{attribute}_i) \quad (7)$$

Where:

- *Weighted Score* is the final-weighted score for the data clump.
- weight_i is the weight assigned to the i -th qualitative factor (e.g., widespread occurrence, size, dependency, consistency with design patterns, degree of code duplication).
- attribute_i is the normalized score (ranging from 0 to 10) for the i -th qualitative factor.

4.3 Implementation

The simple but effective algorithm shown in *Algorithm 1* and described below provides a systematic and customizable approach to prioritize and address the data clumps model smell.

Algorithm 1: Prioritizing data clumps.

Data: Data clumps metrics tuple (δ),
Weights, Thresholds

Result: Prioritized data clumps list

```

1 foreach Data Clump do
2   Normalize scores using (1);
3   Calculate weighted score using
   calculate_weighted_score();
4   Assign priority using
   assign_priority();
5 Sort Data Clumps by Priority (High to Low);
6 Choose Data Clumps for refactoring;
7 Function
   calculate_weighted_score (DataClump,
   Weights):
8   Calculate weighted score using the
   weights provided using (7);
9   return Calculated weighted score;
10 Function assign_priority (DataClump,
   Thresholds):
11   if Weighted Score  $\geq$  Thresholds['high']
   then
12     return "High Priority";
13   else if Weighted Score  $\geq$ 
   Thresholds['medium'] then
14     return "Medium Priority";
15   else
16     return "Low Priority";
```

The algorithm takes as input the metrics of data clumps as defined in the data clumps tuple (δ) in section 4.1, predefined weights and thresholds defined in section 4.2. It outputs a list of prioritized data clumps.

In the main loop of the algorithm (lines 10-14), for each data clump, the algorithm normalizes scores (line 11), calculates weighted scores using the `calculate_weighted_score()` function (line 12) as described in section 4.2. The result is a single numerical value that reflects the importance of each factor based on the specified weights. Based on this value, it assigns priorities using the `assign_priority()` function (line 13). After calculating weighted scores and assigning priorities, the algorithm sorts the data clumps by priority in descending order (high to low) as seen in line 15. Then, the data clumps are selected

for refactoring based on their prioritization (line 16).

4.4 Results and Analysis

The algorithm described above is implemented as a python script. The experiments are run on an Intel Core i7-8550U-1.8 GHz CPU, X64-based PC system running Windows 10.

4.4.1 Factor Contribution Visualization

The algorithm is run with varying sizes of the data clumps. It was observed that the algorithm effectively prioritized data clumps based on calculated weighted scores and assigned priorities, demonstrating its capability to categorize items into *low*, *medium*, and *high* priority levels.

For example, the data clumps factor contribution and priority for an input size of 30 data clumps is visualized in Figure 4. In this plot, each data clump is represented by a horizontal bar in the plot. The factors contributing to the weighted score (widespread occurrence, size, dependency, consistency, degree of code duplication) are colour-coded for easy identification. The total length of each bar corresponds to the total weighted score of a data clump. The bar is segmented into different coloured sections, each representing the contribution of a specific factor to the overall weighted score. Higher segments in the bar indicate that the corresponding factor has a more significant impact on the prioritization of that particular data clump. The length of each coloured segment reflects the proportional contribution of each factor to the overall score. plot provides a visual representation of why certain data clumps are assigned higher priorities. Factors with longer segments contribute more substantially to the overall weighted score, influencing the final priority assignment. Horizontal dashed lines in the plot represent the defined thresholds for low, medium, and high priorities. Bars that cross these lines indicate the priority category of each data clump: *low*, *medium*, or *high*.

Thus, the stacked bar plot reveals insightful patterns in factor contributions, highlighting the significant impact of certain factors on prioritization outcomes.

4.4.2 Scalability

The presented plot in Figure 5 illustrates the scalability evaluation of Algorithm 1 across different sizes of data clumps. The experiment aims to assess the algorithm's performance and efficiency as the size of the dataset varies. This evaluation is crucial for understanding how well the algorithm adapts to increasing

data complexities, providing valuable insights for developers and system architects. The blue line in the plot represents the algorithm's execution time in logarithmic scale concerning the number of data clumps. The logarithmic scale is employed to accommodate a wide range of execution times, allowing for a more comprehensive analysis. The red dashed line signifies the linear scaling reference, serving as a benchmark for comparison. This line illustrates the expected linear scaling behaviour in an ideal scenario.

The algorithm exhibits a positive correlation with data clump sizes, demonstrating scalability as the dataset grows. Execution times remain reasonable, even with a substantial increase in data clump sizes. The logarithmic scaling of execution times provides a clear visualization of the algorithm's efficiency across varying dataset complexities. The provided experimental data includes realistic sizes of data clumps, ranging from 5,000 to 250,000. Corresponding execution times, measured in milliseconds (average of 3 runs), showcase the algorithm's consistent and manageable response to different dataset sizes.

The positive scalability observed in the presented plot indicates that Algorithm 1 effectively handles larger datasets without a disproportionate increase in execution time. This scalability is crucial for real-world applications, where datasets can grow in size over time. The algorithm's performance remains within acceptable limits, offering developers a reliable solution for processing diverse datasets.

4.4.3 Weight Variations

Figure 6 shows radar charts illustrating the impact of weight variations on data clump prioritization. Three subplots in Figure 6 titled *Weights Variation 1*, *Weights Variation 2* and *Weight Variation 3* respectively, represents a different set of attribute weights for the data clumps metrics tuple (cf. section 4.1). The radar charts display different attributes as axes, with the length of each axis corresponding to the normalized attribute values of the data clumps. The charts are colour-filled to represent the priority regions, and the legend includes the weights used in each variation.

The configurations represent different emphasis placed on each factor when calculating the weighted scores for the data clumps. The weights determine the relative importance of each factor in the overall prioritization process. Experimenting with different weight configurations allows us to observe how changes in weights impact the prioritization of data clumps.

The charts in Figure 6 showcase the normalized attribute values of generated data clumps, revealing the influence of weight variations on the prioritization

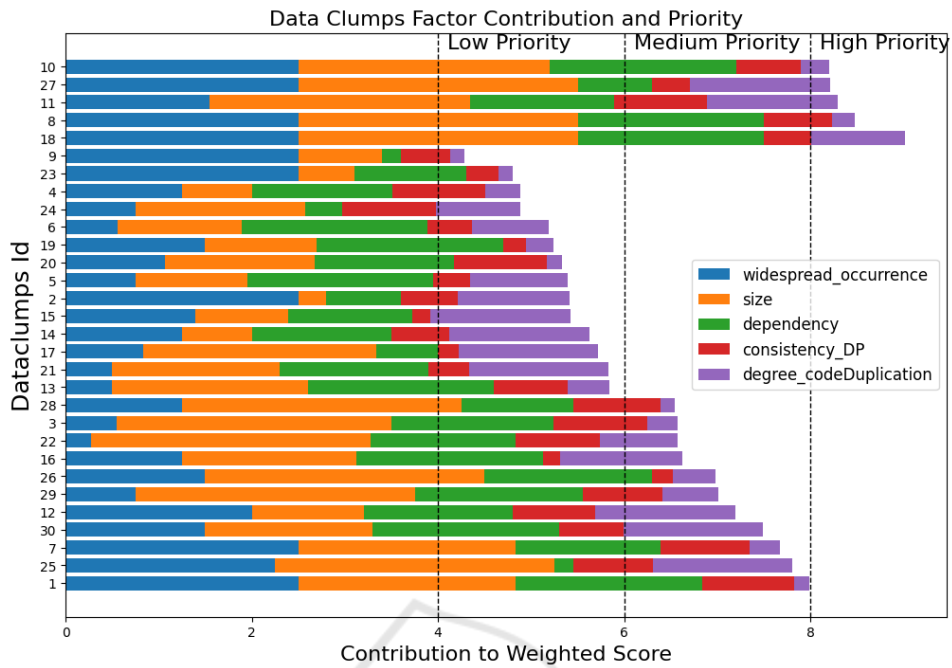


Figure 4: Data clumps factor contribution and priority.

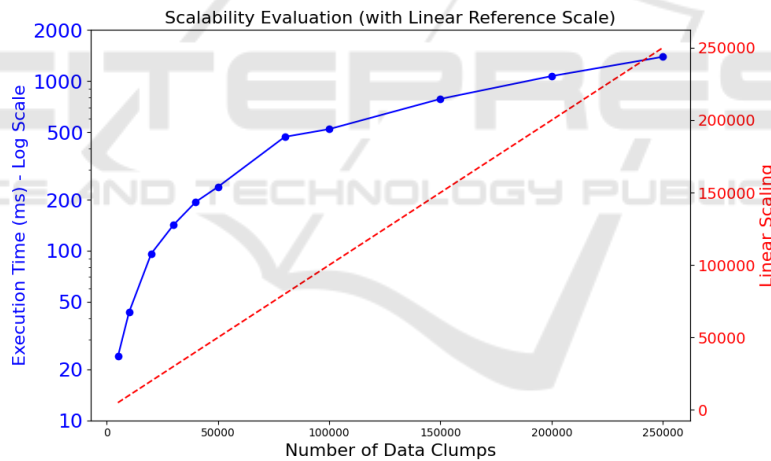


Figure 5: Execution time for various data clumps sizes.

of entities based on calculated weighted scores and predefined thresholds. The experiment involved 30 random data clumps each with metrics WO, SZ, DP, CDP, DC and the application of weight variations to assess the sensitivity of the prioritization algorithm to different weightings. The charts provide developers with a valuable overview, facilitating a better understanding of potential weight adjustments.

4.5 Directions for Enhancements

While the work presented in this paper is a preliminary study, this can be extended in several directions

to enhance the applicability, flexibility, and usability of the metric prioritization approach in various software development contexts.

- **Evaluate on Large Public Datasets:** Apply the approach to diverse and large-scale public datasets representing various types of software systems. Analyse the results to understand the general trends in factor contributions and priorities across different domains.
- **Infer Weights and Thresholds:** Extract insights from the large datasets to suggest default or recommended values for weights and thresholds.

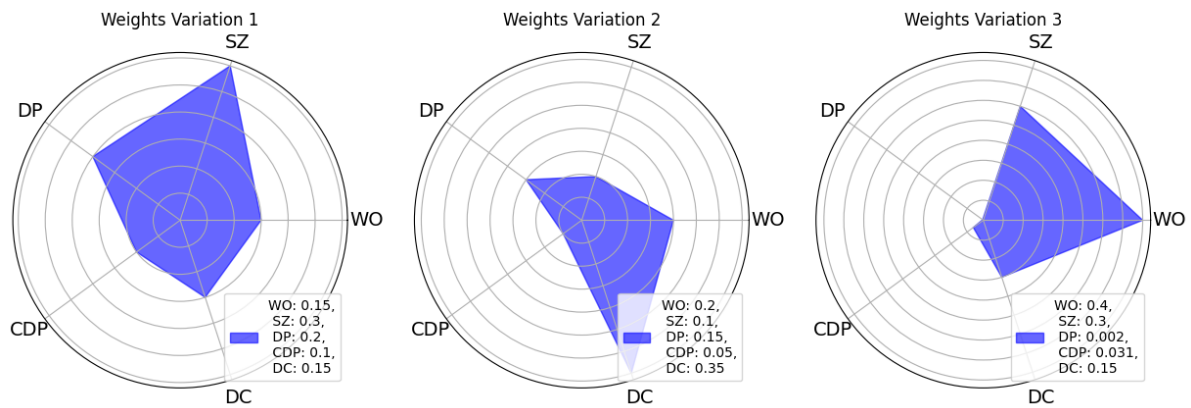


Figure 6: Impact of weight variations on data clump prioritization. Each subplot represents a different set of attribute weights ('WO', 'SZ', 'DP', 'CDP', 'DC').

Consider statistical measures or machine learning techniques to identify patterns and correlations between factors and priorities.

- **Configurability for End Users:** Develop a user-friendly interface allowing end users to customize weights and thresholds based on their specific requirements. Provide guidance or recommendations to users based on the analysis of public datasets, assisting them in making informed decisions.
- **Benchmarking Against Existing Approaches:** Compare the performance and effectiveness of the proposed approach against existing methods of metric prioritization. Conduct benchmarking studies to showcase the strengths and weaknesses of the approach in different scenarios.

5 CONCLUSION

In conclusion, this paper presents a comprehensive approach for prioritizing data clumps in source-code-based and model-based environments. The proposed method emphasizes the importance of addressing data clumps to enhance software quality and sustainability, highlighting the need for systematic prioritization before refactoring. Key contributions of this work include the discussion of both qualitative and quantitative criteria for prioritizing data clumps, and the introduction of a practical, weighted system with threshold-based priority assignment. The approach provides a flexible and customizable solution, enabling developers to tailor their prioritization to their individual and specific project needs.

The experimental evaluation demonstrates the effectiveness of the proposed approach in handling data clumps of varying sizes. The scalability of the al-

gorithm is established through its performance across different dataset sizes, and the impact of weight variations on prioritization outcomes is explored, showcasing the adaptability of the method. Although the evaluation is experimental, it provides a first step towards prioritizing data clumps, yet it still requires deeper analysis.

Future work will focus on extending the approach to evaluate it on large public datasets, infer optimal weights and thresholds, enhance configurability for end-users, and benchmark the approach against existing prioritization methods. By further refining and testing the approach, we aim to contribute to improved code maintenance practices and overall software quality in diverse development environments.

Overall, this work offers valuable insights and a practical guide for software practitioners seeking to prioritize and address data clumps model smell effectively, paving the way for more maintainable, adaptable, and high-quality software systems.

REFERENCES

- AbuHassan, A., Alshayeb, M., and Ghouti, L. (2022). Prioritization of model smell refactoring using a covariance matrix-based adaptive evolution algorithm. *Information and Software Technology*, 146:106875.
- Alkharabsheh, K., Alawadi, S., Ignaim, K., Zanoon, D., Crespo, Y., Manso, M., and Taboada, J. (2022). Prioritization of god class design smell: A multi-criteria based approach. *Journal of King Saud University-Computer and Information Sciences*, 34(10, Part B):9332–9342.
- 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.
- Baumgartner, N. and Pulvermüller, E. (2024). The Life-Cycle of Data Clumps: A Longitudinal Case Study in Open-Source Projects. In *12th International Conference on Model-Based Software and Systems Engineering*, Rome, Italy. Science and Technology Publications, Lda. [Accepted].
- Doan, K.-H. and Gogolla, M. (2019). Quality improvement for uml and ocl models through bad smell and metrics definition. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 774–778.
- Eessaar, E. and Käosaar, E. (2019). On finding model smells based on code smells. In *Software Engineering and Algorithms in Intelligent Systems*, volume 763 of *Advances in Intelligent Systems and Computing*. Springer.
- Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
- Pecorelli, F., Palomba, F., Khomh, F., and De Lucia, A. (2020). Developer-driven code smell prioritization. In *Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20*, page 220–231, New York, NY, USA. Association for Computing Machinery.
- Popoola, S. and Gray, J. (2021). Artifact analysis of smell evolution and maintenance tasks in simulink models. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 817–826.
- 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.
- Zhao, X., Gray, J., and Riché, T. (2021). A survey-based empirical evaluation of bad smells in labview systems models. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 177–188.