FeatureRef: Towards Effective Refactoring of Features

Jose Oliveira
Federal University of Campina Grande
Campina Grande, Brazil
joseglauber@copin.ufcg.edu.br

Thelma E. Colanzi
State University of Maringa
Maringa, Brazil
thelma@din.uem.br

Alessandro Garcia
PUC-Rio
Rio de Janeiro, Brazil
afgarcia@inf.puc-rio.br

Ana Carla Bibiano
PUC-Rio
Rio de Janeiro, RJ, Brazil
abibiano@inf.puc-rio.br

Wesley K. G. Assunção
NC State University
Raleigh, United States
wguezas@ncsu.edu

Rafael de Mello
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil
rafaelmello@ic.ufrj.br

Caio Silva
PUC-Rio
Rio de Janeiro, Brazil
csilva@inf.puc-rio.br

Anderson Uchôa*
Federal University of Ceará
Itapajé, Brazil
andersonuchoa@ufc.br

Anderson Oliveira
PUC-Rio
Rio de Janeiro, Brazil
anderson.jose.so@gmail.com

João Correia
PUC-Rio
Rio de Janeiro, Brazil
jcorreia@inf.puc-rio.br

Everton L. G. Alves
Federal University of Campina Grande
Campina Grande, Brazil
everton@computacao.ufcg.edu.br

Melina Mongiovi
Federal University of Campina Grande
Campina Grande, Brazil

Abstract

Software projects comprise multiple features that capture both functional and non-functional requirements. However, many relevant features emerge over time and, as such, are not modularized into modules (e.g., classes or methods) in the orignal design. Their effective management requires proper modularization through refactorings as software evolves. Poor feature modularization results in design problems (DPs), such as features whose implementations are tangled to each other or scattered across multiple files. Addressing these DPs would benefit from automated recommendations of feature refactoring. However, existing techniques do not explicitly support feature-driven modularization refactoring. Thus, in this paper, we propose *FeatureRef*, an automated technique to assist developers in spotting and refactoring feature-related DPs. *FeatureRef* detects DPs by relying on data from the project design, the source code, and automatically-produced mappings of features in existing program files. The technique uses a topic modeling algorithm to automatically determine how existing features are reified in project files. Then, *FeatureRef* combines feature mappings with indicators of poor structural code quality to find DPs. Finally, to create refactoring recommendations, *FeatureRef* relies on a novel refactoring recommendation strategy, which combines refactoring heuristics with search-based optimization. We evaluated *FeatureRef* through an empirical study with six open-source projects. The results show that our approach removes nearly three times more feature-related DPs compared to a state-of-the-art baseline that strictly relies on structural metrics to guide refactoring of feature-related code smells, such as God Class (feature tangling) and Feature Envy (feature scattering).

Keywords

Software redesign, design problems, search-based optimization, feature modularization

1 Introduction

Software projects commonly comprise multiple features, representing functional and non-functional requirements [13]. In many projects, features emerging at design or evolution time are the main unit of work distribution [46]. Thus, a fundamental design practice is supporting the proper modularization of each feature in a project [5, 44]. Despite the importance of feature modularization, engineers often introduce Design Problems (DPs) by making decisions that negatively affect feature modularity [27, 28]. Feature-related design problems comprise scattered features (i.e., a feature logic spread across multiple program elements) [5, 19, 39] and feature overloads (i.e., multiple unrelated features implemented in single program elements) [19, 39]. These problems hinder maintenance of feature code as features' responsibilities are tangled or spread in multiple elements.

The effectiveness of existing techniques (e.g., [39, 42, 51]) for identifying these feature-related DPs is unknown. They are often strictly based on internal code measures and detection of code smells. When identified, DPs should be removed through code refactorings [18]. However, deciding where and how to refactor feature-related DPs is far from trivial. Traditional code measures and smells are not enough for properly identifying feature-related DPs [39, 42, 51]. Moreover, even when the locations of DPs are known, refactorings performed in practice may be unable to completely remove them [9, 40]. The core deficiency is that existing work does not take into consideration automatically-generated feature representations.

There are three major challenges related to detecting and removing feature-related DPs. First, feature-related DPs often manifest across multiple elements of a system, making their detection difficult. This occurs because DPs involve different types of symptoms that must be considered together with feature-to-code mappings for effective identification, such as scattered method calls or duplicated conditional logic (all pertaining to a single feature), or unexpected feature interactions [38–40, 51]. Second, developers often need help

*Also with North Carolina State University, Raleigh, United States.

understanding all the multiple symptoms and causes of a feature-related DP, before they can fix it, which makes the process time-consuming and error-prone if not done carefully [39, 51]. Otherwise, negative consequences of feature DPs may persist. Third, developers need support in deciding which refactorings to apply and how, as removing a feature-related DP often requires a sequence of multiple non-localized refactorings. These refactorings may also involve elements without clear syntactic dependencies [12, 38, 50], making them difficult to identify.

Despite existing approaches for detecting and removing DPs (Section 2), they fall short in addressing the challenges described above. In particular, some techniques recommend refactorings targeting code-level smells related to poor feature modularization, such as God Class (representing dense feature tangling) and Feature Envy (representing mislocated functionality). However, these approaches rely exclusively on structural properties of classes and methods, without explicitly modeling how features are implemented and distributed across the system. A single feature may contain multiple feature envies and other smells, which should be better refactored once. Certain parts of a feature many not contain a documented smell and are, therefore, ignored by the techniques. As a result, they are limited in their ability to accurately identify and refactor feature-related DPs that span multiple elements and require semantic understanding of functionality.

Recent advances in refactoring recommendation have explored the use of deep learning and transformer-based models to detect and suggest refactorings, including approaches such as ROSE [36]. These techniques leverage structural and historical information to identify architectural and design issues. However, they typically do not explicitly model how features are represented and distributed across the system, which limits their applicability to feature modularization problems. Despite complex feature-related refactorings being reported in the literature [8–11, 50], there is evidence that many refactorings performed in practice are not effective [7, 12, 47]. That happens because developers usually (1) perform incomplete refactorings [7–9], and (2) select combinations of refactorings that are not effective for major feature-wide DPs [1, 47], such as feature scattering and overload. As a result, feature-related DPs may not be completely removed, or, sometimes, design quality can get worsened [12, 50].

To address these limitations, we propose a new recommendation technique called *FeatureRef*. The technique uses as input not only the software source code, but also its project design. From the project design, *FeatureRef* uses a topic modeling [49] algorithm to extract the distribution of features across the project elements. We have not used LLMs to capture feature-to-code representations as recent evidence suggests that LLMs is still struggle [54] with the reliability and precision required for high-level architectural feature mappings. While LLMs can handle structural changes, their ability to consistently and accurately relate abstract requirements to distributed source code elements remains a challenge. Also, *FeatureRef* uses the source code to collect quality measures and code smells, relating them to the relevant features discovered with the topic modeling algorithm. To provide effective refactorings, we have designed a new heuristic for refactoring recommendations based on feature modularity. *FeatureRef* combines our new heuristic with two heuristics inspired by the state-of-the-art techniques for recommending effective refactorings. The recommendation relies on search-based optimization to create and incrementally evolve (*i.e.*, improve) multiple possible recommendations. Finally, to improve efficiency, we defined a strategy for selecting which elements to refactor based on the number of feature-related DP symptoms.

We implemented a reference tool to evaluate *FeatureRef* and conducted an empirical study involving six open-source projects. For the search-based optimization, we selected two algorithms, namely NSGA-II [16] and MOSA [53], which presented promising results in refactoring recommendation studies [2, 24]. The evaluation is based on both quantitative and qualitative analysis. The results show that *FeatureRef* outperforms the approach proposed by [38], used as a BASELINE in our evaluation. Specifically, *FeatureRef* leads to a greater reduction in the number of code smells, lowers the number of features per code element, and improves cohesion by better grouping related functionality within system components.

In summary, the main contributions of our work are: (1) proposal of a refactoring recommendation technique that combines search-based optimization and topic modeling to overcome the three challenges above; (2) implementation of a reference tool; and (3) an empirical study that reports new promising findings on the proposed technique.

2 Background and Search-based Refactorings

Composite Refactoring. Complex design problems often require sequences of refactorings rather than isolated transformations. Existing tools and techniques for refactoring detection focus mainly on individual refactorings [34]. To overcome this limitation, optimization techniques have been applied to suggest effective sets of refactorings. Approaches based on genetic algorithms, local search, and simulated annealing aim to find combinations of code changes that optimize structural quality metrics in a more general way [21]. Inspired by these advances, this work explores the generation of *composite refactorings*, sequences of changes, to enhance code structure in a more comprehensive way, addressing multiple quality aspects simultaneously. In this context, optimization becomes a key element [8], [50], [9].

A Baseline Technique for Recommending Composite Refactorings. Oizumi et al. proposed an approach to recommend composite refactorings [38]. They also demonstrated that heuristics based on code smells and internal quality attributes can effectively guide the removal of major structural problems in software systems. Hereafter called the BASELINE technique, it systematically identifies composite refactoring patterns and uses them to construct heuristics to eliminate common types of feature-related code smells such as Complex Class, Feature Envy, and God Class. Unlike BASELINE, which only analyzes source code, *FeatureRef* incorporates explicit information related to features into the refactoring process.

Recent approaches have explored the use of machine learning and transformer-based models for refactoring recommendation, such as ROSE (ESEM 2025), which focuses on architectural smells using deep representations of code structure. While these approaches have shown promising results, they rely primarily on structural and historical information and do not explicitly capture how features are modularized across the system, not explicitly addressing feature-related DPs. The BASELINE technique was selected because

it relies on structural heuristics similar to those used in practice for detecting code smells related to poor modularization. Although more recent approaches exist, they are not directly applicable to our problem, as they do not incorporate an explicit representation of features in the code.

Optimization Techniques. Recent studies have shown that approaches based on multi-objective evolutionary algorithms perform well when solving a variety of problems, especially because these algorithms can handle multiple criteria that are part of how the problem itself is defined [14, 16]. Among the algorithms most commonly used for this type of problem are NSGA-II and MOSA. NSGA-II organizes candidate solutions into Pareto fronts based on dominance relationships and uses a crowding distance measure to maintain diversity across solutions [16]. In contrast, MOSA was designed to efficiently handle many-objective problems (three or more conflicting objectives), employing a coverage-based sorting strategy and maintaining an external archive to ensure diversity and convergence even in high-dimensional spaces [23] [33].

3 The FeatureRef Approach

FeatureRef aims to support refactoring for feature modularization problems. It was designed based on previous studies [42], [41], [40] and [38]. Figure 1 presents an overview of the technique and its execution steps, implemented in the tool[1]. It is important to note that FeatureRef is a refactoring recommender; it identifies and suggests sequences of transformations but does not automatically execute them in the source code or validate the resulting state. A pseudo-code representation of the FeatureRef algorithmic process is available in the project's official repository.

To illustrate the use of *FeatureRef*, Figure 2 shows an example from the OkHttp repository[2] , where *FeatureRef* generates a refactoring recommendation for the overloaded class *Transmitter*. The technique identifies Feature Overload based on symptoms such as Large Class, God Class, Complex Class, Feature Envy, and Dispersed Coupling, as well as the presence of multiple features within the same class. This analysis, *FeatureRef* recommends splitting the class into two, improving feature modularization. The selected solution achieves better cohesion, quality metrics, symptom reduction, and fewer refactorings compared to alternative solutions.

3.1 Feature-related Collection

FeatureRef can detect DPs and recommend refactorings based on data on feature modularization. This step aims to collect information from the source code to infer the project's functionalities using topic modeling techniques. It was designed to address *Challenge 1* (presented in the introduction), based on evidence from the literature pointing out that feature-related DP identification requires heterogeneous symptoms [51, 52].

The procedure to detect features and select the code elements for the identification of DPs is composed of four steps, namely *Element selection, Topic Model Construction, Feature Identification and Association*, as follows.

[1]https://github.com/opus-research/feature-ref-package
[2]https://github.com/square/okhttp

Element Selection. *FeatureRef* allows for the definition of a strategy for selecting the elements to analyze and generate refactoring recommendations. Our selection strategy is defined as the *top ten code elements with a higher chance of being impacted by DPs*. For instrumentation, we decided to rely on the number of DP symptoms. Thus, this heuristic is expected to select the elements more likely to need feature modularization. The pre-selection of files is performed based on the symptoms initially detected. The metrics used in this process are: number of fields, number of methods, and number of statements. Figure 2 shows the values of these metrics for the Transmitter.java file. These results suggest that this class contains many attributes, methods, and statements, indicating that it likely handles multiple responsibilities.

Topic Modeling. *FeatureRef* relies on a Topic Modeling algorithm for finding features, receiving as input the top degraded element collected in the previous steps. For this task, we use the Machine Learning for Language Toolkit (Mallet) [32]. We use their Latent Dirichlet Allocation (LDA) parallel topic modeling implementation, previously applied in previous studies in the field [49]. A topic modeling algorithm relies on textual data to identify recurrent topics based on their occurrence in the analyzed documents. In our case, the documents are the source code files.

Figure 2 illustrates the Topic Modeling process. For the problematic *Transmitter* Java class, topic modeling was performed to identify the most representative topics within its textual content. Two topics, 21 and 44, were identified, each consisting of frequently occurring words in the file and represented as a set of tokens.

Feature Identification and Association. Features are inferred from topics and associated with code elements based on token frequency (Figure 2). Each set of tokens forming a topic is inferred to represent a possible functionality derived from the topic's content. These functionalities may be associated with specific code elements, and identifying these associations is key to detecting problematic code elements related to particular functionalities.

After the identification of features, *FeatureRef* assigns these features to the source code Types and Methods. Each association attempt is evaluated based on how frequently the topic's tokens appear within the textual code element. The more dominant the tokens are, the stronger the indication that the functionality is associated with that particular code element. This may suggest that the element is handling too many responsibilities, making it a potential candidate for refactoring. This leads *FeatureRef* to investigate this class, which contains a DP. The output is a more refined set of code segments with potential design issues, helping us to recommend refactorings that are more related to the system functionalities.

3.2 Quality Collection

Given the code elements provided by the *Feature-related Collection* step, *FeatureRef* conducts a quality assessment based on the combination of multiple feature-related DP symptoms, addressing *Challenge 2*. *FeatureRef* allows the use of various types of symptoms. We selected three types of symptoms, namely *Internal Quality Measures*, *Code Smells*, and *Feature-based Symptoms*. Based on the information about symptoms, a composite refactoring recommendation is generated. Then, optimization algorithms are applied to

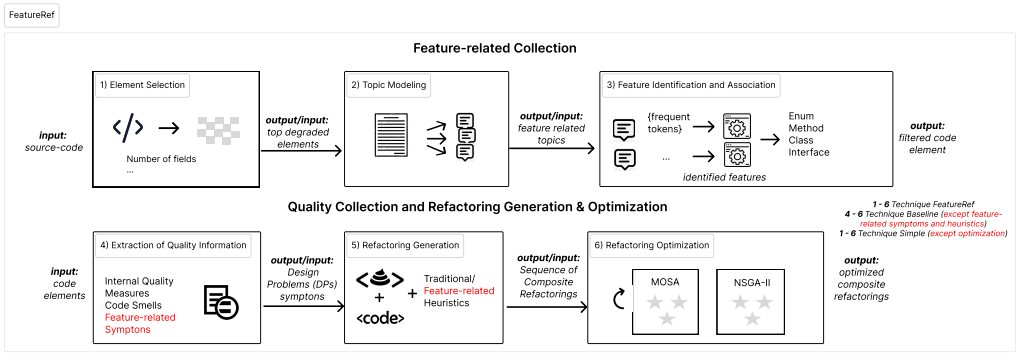


Figure 1: Overview of the *FeatureRef* technique

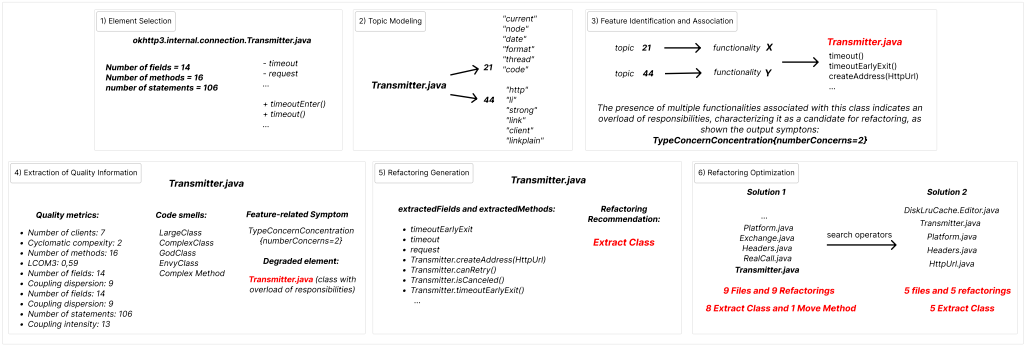


Figure 2: Example of all steps of Refactoring Solution for the Okhttp Project

refine these recommendations, considering that topic modeling may produce diverse results due to the size of the search space. The steps involved in this process are detailed in the following subsections.

Extraction of Quality Information. Based on the code elements identified during the Feature-related Collection step, this stage focuses on gathering evidence of potential design issues within those elements. To accomplish this, Internal Quality Metrics, Code Smells, and Feature-based Symptoms are analyzed. After applying the Feature-related step, *FeatureRef* can now operate on specific elements that may be associated with design flaws. Figure 2 presents the quality metrics collected for the identified class, along with the associated code smells and feature-related symptoms. We describe the three types of symptoms as follows.

Internal Quality Measures comprise source code metrics relevant to feature modularization. To detect *Feature Overload* and *Scattered Feature*, we consider measures such as *Coupling Dispersion, Coupling Intensity, Lack of Cohesion, Cyclomatic Complexity, Number of Clients, Methods, Fields, and Statements.*

These metrics capture different structural aspects affected by such problems. In particular, coupling and cohesion are established indicators of poor modularization [26], while size-related measures provide indirect evidence of degradation. In our example, high values - especially in the number of methods - suggest an overload of responsibilities in the analyzed class.

Code Smells are surface indicators of design problems in source code [18]. In *FeatureRef*, we use rule-based heuristics [31], derived from internal quality measures, as commonly adopted by state-of-the-art approaches. We focus on smells related to feature modularization [39], namely Complex Class, Dispersed Coupling, Feature Envy, God Class, Large Class, and Lazy Class.

Table 1 illustrates how internal quality metrics support smell detection. In our example, high numbers of methods, fields, and statements indicate excessive responsibilities (Large Class), while moderate cohesion and high coupling suggest complexity (Complex Class). Moreover, the presence of multiple clients points to centralized functionality, indicating a potential God Class.

Table 1: Code Quality Metrics and Related Symptoms

Metric	Indication	Smell
#clients = 7	High usage	GodClass
#methods = 16	Many methods	LargeClass
lcom3 = 0.59	Moderate cohesion	GodClass
#fields = 14	Many attributes	Large, Complex
coup_disp = 9	External access	ComplexClass
#statem = 106	Long code	LargeClass
coup_int = 13	High dependency	ComplexClass

Feature-based Symptoms capture issues related to poor feature modularization. We define two heuristics based on feature detection: *Feature Concentration*, which identifies classes with an

above-median number of features as candidates for *Feature Overload*, and *Feature Dispersion*, which detects features spread across multiple non-dominant classes, indicating scattering. These heuristics are directly aligned with the targeted DPs.

In our example, the class *Transmitter.java* is identified with the symptom *TypeConcernConcentration* (numberOfConcerns = 2), indicating multiple features concentrated in a single class. This reflects *Feature Overload*, as the class exceeds the average number of features in the project and accumulates multiple responsibilities, violating feature modularization principles.

3.3 Refactoring Generation

This step generates the refactorings to improve the feature modularization of a subset of code elements. As we can see in Figure 1, once the design symptoms have been identified in the code elements, *FeatureRef* uses these design issues alongside both traditional heuristics and feature-related heuristics. Although the selection of heuristics is flexible, we selected the following initial set of heuristics for this study:

Move Method Heuristic. *FeatureRef* finds the suitable class for moving a method by counting the number of calls and accesses to fields. Methods are moved to the class with the highest structural affinity. The rationale behind this heuristic is that a class with the highest bound factor can be the best candidate for cohesively receiving the moved method [38].

Extract Class Heuristic. For generating Extract Class refactorings, *FeatureRef* uses a heuristic inspired by the one proposed by Fokaefs et al. [17]. *FeatureRef* uses Agglomerative Hierarchical Clustering to find groups of fields and methods that can be extracted. Itc starts by including each method and field in separate clusters. Then, the clusters are compared and merged based on the Jaccard distance metric. This process continues through multiple iterations until no clusters can be merged. As a result, our heuristic recommends extracting the smaller clusters of fields and methods.

Feature-based Move Element Heuristic. This heuristic is inspired by the *MethodBook* technique proposed by Bavota et al. [6], which relies on topic modeling to find Move Method candidates. Similarly to them, *FeatureRef* identifies multiple candidates for receiving a method based on feature similarity. In this case, we include only as candidates the classes for which the predominant feature is the same as the method. The heuristic generates the Move Method targeting a randomly selected class among the available candidates. The other candidates are stored to be used later in the refactoring optimization process.

Feature-based Extract Class Heuristic. *FeatureRef* includes a heuristic for generating Extract Class refactorings based on features. It selects the classes that contain two or more features and recommends the extraction of methods implementing the non-predominant features of the class. Besides the methods, it selects the fields used only by the extracted methods.

These four heuristics were selected for their effectiveness in supporting refactorings that address feature-related DPs. As shown by [38], [17], and [6], such techniques improve modularity by restructuring code based on cohesion, coupling, and feature similarity, helping to reorganize scattered or misplaced functionalities. Although other refactorings could also improve feature modularization, this study focuses on those targeting the internal structure of classes. In our example, the identified symptoms highlight critical elements of the class, particularly a cluster of closely related methods and fields, suggesting an embedded responsibility and a tendency toward a God Class.

Based on this analysis, *FeatureRef* recommends an Extract Class refactoring (Figure 2) to isolate this cohesive subset into a new class. This recommendation follows the defined heuristics and is part of a broader refactoring sequence aimed at improving modularity and maintainability. The generated recommendations are abstract refactoring plans requiring manual implementation, without automated verification of compilation, test execution, or behavioral preservation.

3.4 Refactoring Optimization

FeatureRef employs a search-based algorithm [22] to recommend refactorings that improve code structure according to defined optimization objectives, addressing *Challenge 3*. Unlike traditional approaches (e.g., MORE [43], RefBot [3], and others surveyed in [30]) that rely on random initial populations, *FeatureRef* constructs its initial population by combining refactorings derived from four predefined heuristics, enabling the search to start from higher-quality solutions and potentially reducing optimization effort.

The optimization problem is modeled as a refactoring sequence vector, where each position represents the refactorings applied to a specific element, which may receive multiple or no refactorings. As illustrated in Figure 3, this representation allows *FeatureRef* to focus the optimization on selected elements, improving the effectiveness of the search.

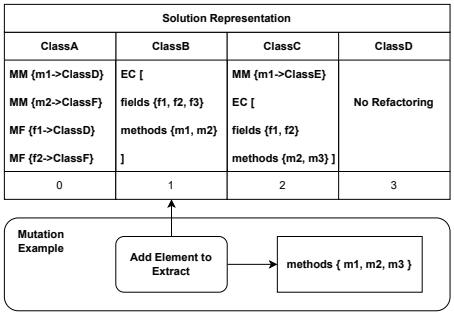


Figure 3: Example of solution representation

The choice of operators and objectives in *FeatureRef* aims to efficiently generate high-quality refactoring solutions, following the guidelines from [24], focused on reducing smells and improving modularization. The approach includes four operators: *Replace Refactoring Type*, which changes the refactoring applied to an element; *Add Refactoring to an Element*, which complements existing refactorings; *Remove Refactoring*, which removes part or all of a refactoring; and *Change Move Target*, which redirects a Move Method to another candidate element.

To evaluate solutions, *FeatureRef* defines five minimization objective functions. The **Quality Measures** objective aggregates structural metrics, as in prior work (e.g., [2]), while **Lack of Cohesion** is treated separately due to its direct relation to feature modularization. The **Density of Symptoms** follows [24], measuring the ratio of symptoms after and before refactoring. The **Number of**

Features per Element evaluates how solutions improve feature distribution, and the **Number of Refactorings** favors solutions with fewer transformations.

Overall, these operators and objectives enable *FeatureRef* to explore and refine refactoring solutions, improving cohesion, reducing coupling and design symptoms, and optimizing the number of applied refactorings.

The application of search operators enables *FeatureRef* to identify multiple refactoring solutions with different scopes and trade-offs. For example, as shown in Figure 2, one solution applies nine refactorings across nine files to maximize modularization, while another, more focused, uses only five refactorings in five files, still achieving significant improvements in cohesion and complexity. This demonstrates that both broad and targeted strategies can be effective. Through crossover and mutation operators, *FeatureRef* not only automates refactoring discovery but also explores and evolves alternative solutions, allowing developers to balance goals such as structural improvement and number of changes. This gives developers more flexibility and control when deciding how to refactor their systems.

FeatureRef is built on the JMetal framework[3], which supports multiple optimization algorithms and extensibility. In this study, we adopt MOSA [53] and NSGA-II [16] as refactoring recommendation strategies. MOSA showed good performance in preliminary evaluations and in related work [24], while NSGA-II is widely used in software engineering optimization [15]. In NSGA-II, the proposed operators act as mutation mechanisms, and crossover combines elements from different solutions to generate new ones. Although these algorithms were selected, further studies may explore additional optimization techniques.

Finally, *FeatureRef* includes baseline strategies for comparison. The SIMPLE strategy generates five solutions: four based on individual heuristics and one combining them randomly. The BASELINE strategy follows the same structure but replaces feature-driven heuristics with traditional ones, enabling comparison with existing approaches.

4 Study Design

We designed a multi-case study to evaluate the proposed technique. Our study goal is *to evaluate the effectiveness of the FeatureRef's refactoring recommendations to improve feature modularization.* To guide our study, we address the following research questions (RQs).

RQ_1: *What is the effectiveness of feature-based strategies when compared to a baseline?* This RQ aims to evaluate whether our refactoring heuristics, in combination with search-based optimization, can improve the recommendation of refactorings for feature modularization. To answer RQ_1, we evaluated the effectiveness of search-based algorithms in generating refactoring recommendations compared to a baseline (see Section 3.4). This evaluation was performed based on the impact of solutions on (1) the number of code smells, (2) the number of features per element, (3) the lack of cohesion of refactored elements, and (4) their computing time.

RQ_2: *What is the impact of solutions generated by FeatureRef on feature modularization?* To answer RQ_2, we compared the solutions generated by each of the strategies described in Section 3.4. The idea behind RQ_2 is to understand how the use of feature-based heuristics and search-based algorithms impacts feature modularization. Furthermore, we did a manual evaluation to identify if the technique's components were able to improve feature modularization from the developers' point of view.

4.1 Target Software Projects

We selected a set of six industry-strength open-source projects for our study. We selected projects following five criteria: (1) implemented in the Java programming language; (2) use pull request reviews as a mechanism to receive and evaluate contributions; (3) have at least 1,000 commits; (4) are at least five years old, and (5) are currently active. We understand that exploring this diversity is important to our study because it allows us to evaluate the proposed technique over a heterogeneous set of features and design decisions that may influence the required refactorings. Table 2 provides more details about the six selected projects.

Table 2: Overview of the Target Projects

Project	Domain	Commits	PRs	Release
Dubbo	RPC framework	5356	4702	3.0.6
Fresco	Image library	3150	407	2.6.0
Jenkins	CI server	31686	6351	2.319.3
OkHttp	HTTP client	5072	3457	3.14.0
RxJava	Async library	5969	3678	3.1.0
Spring Security	Security library	10142	1865	5.6.0

4.2 Experimental Configurations

Detailed information about the experimental setup, including algorithm configurations, parameters, and execution settings, is available in the README of project repository[4].

4.3 Data Collection Procedures

We followed the procedures described below.

Topic modeling training: Using the configurations described in the previous section, we ran the Mallet tool for training the topic models. Since the target projects are from different domains, we created a model per project. The input is composed of the source code files of the target project, and the result is the topic model. In the next step, we use the topic model of each project in *FeatureRef* to identify features based on topic inference.

***FeatureRef* execution.** We implemented all the refactoring recommendation strategies (MOSA, NSGA-II, and SIMPLE, all using LDA) in the same tool. *FeatureRef* contains both the baseline and the feature-driven strategies, which allows us to perform comparisons based on the same quality measures. For each recommendation strategy, *FeatureRef* provides the list of solutions with their respective recommended refactorings and fitness values. Moreover, it provides information about the design quality before and after performing the recommended refactorings. For each target project, we ran *FeatureRef* and collected the aforementioned results for 30 executions of each recommendation strategy evaluated in this study.

[3]https://github.com/jMetal/jMetal

[4]https://github.com/opus-research/feature-ref-package

4.4 Quantitative and Qualitative Analysis

To answer our RQ_s, we defined three key data analysis procedures.

Selection of the best solutions. We selected the non-dominated solutions [14] across the 30 executions. The *non-dominated solutions* are those reaching the best fitness to all objectives.

Comparing Strategies. We relied on multiple quality impact criteria to compare MOSA, NSGA-II, and SIMPLE across all projects: (1) number of symptoms, (2) number of features per element, and (3) lack of cohesion. Such criteria were used to evaluate the generated solutions based on the differences before and after applying the recommended refactorings. Also, we performed a quantitative analysis based on descriptive statistics and the Mann-Whitney U test for all criteria.

Recommendations Analysis. We manually analyzed a representative subset of recommendations to complement the quantitative evaluation. To reduce the cognitive effort required to understand different codebases, we focused on recommendations from the OkHttp project. We assessed the technical soundness, practical feasibility, and impact of the recommended refactorings on both structural quality and feature modularization, considering design symptoms and feature distribution across software elements.

5 Results and Discussion

This section presents the results of our evaluation of *FeatureRef*.

5.1 On the Quality Impact of Feature-driven Strategies

To answer RQ_1, we evaluated four refactoring recommendation strategies, comparing feature-driven approaches with a rule-based baseline. Table 3 summarizes the median results for three quality criteria computed over the top 10 degraded elements of each project. Negative values indicate improvements after refactoring, and the best results are highlighted in bold.

DP Symptoms. SIMPLE and NSGA-II achieved the best results in most projects, with MOSA presenting similar results except for Dubbo. BASELINE showed relevant improvement only for Jenkins. A Mann-Whitney U Test indicates, with 95% confidence, that SIMPLE and NSGA-II significantly outperform BASELINE and MOSA. **Number of Features per Element.** Differences were smaller across strategies, but NSGA-II achieved the best results in four projects. BASELINE performed worst in all cases, even increasing features per element in OkHttp.

Lack of Cohesion. NSGA-II consistently achieved the best results across all projects. SIMPLE and MOSA were less competitive in this criterion compared to previous ones. **Coupling Measures.** MOSA achieved the best reduction in coupling intensity, while NSGA-II outperformed all strategies in coupling dispersion, reinforcing its effectiveness for feature modularization.

> **Finding 1.** NSGA-II achieved the best overall results, consistently reducing symptoms and lack of cohesion across projects, while also providing the highest reduction in features per element in most cases.

Computing Time. BASELINE and SIMPLE required less than 30 seconds. In contrast, MOSA often required around one hour, reaching 1h14min for Dubbo. NSGA-II was significantly faster, ranging from under 4 minutes to about 16 minutes. Although slower than BASELINE and SIMPLE, its execution time is more practical and justified by its superior results.

> **Finding 2.** While the computing time is still not ideal, the NSGA-II strategy can be viable give its consistently positive impact.

5.2 Fitness of Generated Solutions

For answering RQ_2, we analyze how the solutions obtained by each strategy are dispersed in the solution space regarding our objective functions: Density of Symptoms (Density), Number of Features per Element (NFEAT), and Number of Refactorings (NREF). The first two objectives are direct indicators of the impact of feature modularization, while the last provides an overview of the required effort for applying each solution in practice.

Solution Space for Fresco. The solutions generated by NSGA-II are close to those generated by SIMPLE, but when focusing on the best solutions, NSGA-II is significantly better: only NSGA-II solutions achieve the simultaneous minimization of Density and NFEAT, doing so with fewer refactorings than SIMPLE. Compared to MOSA, NSGA-II explored the search space better, resulting in higher solution diversity — a consequence of using crossover and mutation operators [20]. The BASELINE solutions presented the best NREF results, but their DENSITY and NFEAT results were among the worst.

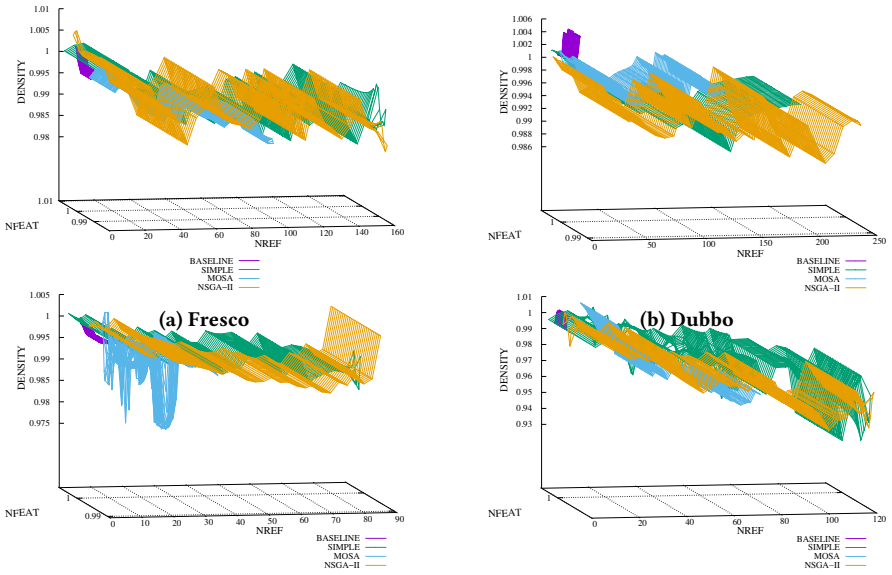
Solution Space for Dubbo. Results are similar to Fresco, with the main difference being that the BASELINE strategy performed significantly worse than all others, even when considering only solutions with smaller NREF, all other strategies presented better DENSITY and NFEAT. Most MOSA solutions also performed worse than NSGA-II regarding DENSITY.

Solution Space for Jenkins and RxJava. For Jenkins, all strategies produced similar solutions, closer than in other projects. MOSA achieved better DENSITY results while keeping NREF below 40 in most cases, whereas NSGA-II obtained better NFEAT results but required more than 50 refactorings. We attribute this behavior to the quality of the topic model, which affected feature-based heuristics. For RxJava, MOSA also performed better in terms of DENSITY. This was the only project where BASELINE generated more non-dominated solutions, although its solutions became worse than the others for DENSITY and NFEAT as NREF increased.

Solution Space for OkHttp. MOSA, NSGA-II, and SIMPLE solutions are very close for the first half of the NREF axis, with MOSA performing slightly better on DENSITY. As the number of refactorings increases, NSGA-II tends to improve its impact on DENSITY and NFEAT, although a larger number of refactorings may make such solutions impractical. To summarize, NSGA-II explores a larger search space than the other strategies, providing more opportunities to find solutions that significantly improve feature modularization. When considering only solutions with smaller NREF values, NSGA-II outperforms the others on either DENSITY or NFEAT. While MOSA can present better DENSITY results when

Table 3: Median Quality Impact of Non-Dominated Solutions on the Top 10 Degraded Elements

	# Smell Difference				# Features Difference				LCOM Difference			
Project	BASE	SIMP	MOSA	NSGA	BASE	SIMP	MOSA	NSGA	BASE	SIMP	MOSA	NSGA
Fresco	-6	**-23**	-16	**-23**	-1.5	**-7**	-4	**-7**	0.0468	-0.2043	-0.0723	**-0.2212**
RxJava	-15	-35	-13	**-37**	-2	-10	-5.5	**-11**	-0.0544	0.0151	0.0415	**-0.0747**
Jenkins	-24	-27	-21	**-29**	-4	-5	-5	**-6**	-0.0261	-0.0923	-0.0921	**-0.1378**
Spring Security	-5	**-19**	-17	**-19**	0	**-5**	**-5**	-4	0.0007	-0.0913	-0.0469	**-0.1145**
Dubbo	-11	**-24**	-11	-23	-1	**-2**	**-2**	-1	0.1247	-0.1452	-0.0475	**-0.1972**
OkHttp	-6.5	-26	-20	**-27**	1	-3	**-4**	**-4**	-0.0741	-0.2193	-0.1549	**-0.2541**
All	-10.5	**-26**	-16	**-26**	-2	-5	-4	**-6**	-0.0202	-0.114	-0.066	**-0.151**



(c) Jenkins **(d) OkHttp**

Figure 4: Partial view of the solution space for all projects

Table 4: Median Computing Time (in Minutes) for Generating Solutions

Project	BASE	SIMP	MOSA	NSGA-II
Fresco	0.0478	0.0766	22.1735	3.9935
RxJava	0.1502	0.1879	63.1804	14.7194
Jenkins	0.1438	0.1978	64.5497	12.1198
Spring Security	0.1139	0.1576	58.7528	13.3737
Dubbo	0.1622	0.2133	74.6844	15.9935
OkHttp	0.0508	0.1474	21.0723	3.2373
All	**0.1288**	**0.1774**	**62.2587**	**12.4134**

restricting solutions to fewer refactorings, NSGA-II tends to be superior on NFEAT, leading us to conclude that it is the best strategy for achieving *FeatureRef*'s goal.

Finding 3. NSGA-II better fits the *FeatureRef*'s goal. NSGA-II explores a larger search space, generating solutions (refactoring recommendations) that improve feature modularization with a smaller number of refactorings.

5.3 Best Solutions Analysis

Since NSGA-II and MOSA presented the best results, we now analyze their best solutions, selected based on the *Euclidean Distance* from the ideal solution. Table 5 shows the fitness values before (original) and after (refactored) applying the best solution for each project and strategy. For the same project, some original fitness values vary between MOSA and NSGA-II due to the non-deterministic nature of the topic modeling algorithm.

High impact on the DENSITY and NFEAT objectives. For all target projects, both DENSITY and NFEAT were consistently improved after applying the best solutions, showing that our refactoring heuristics and mutation operators acted directly on feature modularization, even without applying any weight to the objective functions.

Negative impact on the LCOM objective. Fitness values worsened in only a few cases (highlighted in red in Table 5), with LCOM being the most negatively impacted objective: five cases with MOSA and three with NSGA-II. We believe that effective LCOM improvement may require refactoring types beyond those currently supported by *FeatureRef*, such as Extract Method. A possible negative

Table 5: Fitness Values Before (Original) and After (Refactored) Applying the Best Solution

		DENSITY		NFEAT		LCOM	
Project		MOSA	NSGA	MOSA	NSGA	MOSA	NSGA
Fresco	Original	3.445	3.384	1.804	1.773	0.565	0.565
	Refactored	0.995	1.000	1.001	1.007	**0.567**	**0.566**
RxJava	Original	3.580	3.121	1.988	2.032	1.304	1.304
	Refactored	0.995	0.999	0.998	1.001	1.302	1.300
Jenkins	Original	2.990	3.073	1.484	1.533	0.485	0.485
	Refactored	0.981	0.998	1.000	1.001	**0.487**	0.485
Spring Sec.	Original	3.224	3.224	1.674	1.654	0.551	0.551
	Refactored	0.998	0.998	1.000	1.000	**0.552**	**0.552**
Dubbo	Original	3.218	3.297	1.753	1.790	0.540	0.540
	Refactored	1.000	0.999	1.002	1.004	**0.542**	**0.545**
OkHttp	Original	3.581	3.581	1.725	1.725	0.676	0.676
	Refactored	1.007	0.981	1.013	1.017	**0.682**	0.668

correlation between LCOM and DENSITY or NFEAT objectives is another hypothesis that deserves further investigation.

Low to no impact on Quality Measures. In most cases, QMEASURES remained unchanged while DENSITY improved for all projects. We conjecture that this lack of correlation stems from our use of feature-based symptoms: our feature-driven heuristics significantly impact feature-based symptoms, while their effect on traditional symptoms (captured by QMEASURES) is comparatively small. For clarity and conciseness, we removed the QMEASURES column from the table, as it showed negligible variation and did not affect the interpretation of our results.

Number of Refactorings. Table 6 shows the number of refactorings for the best solutions.

Table 6: Number of Refactorings

Project	MOSA	NSGA-II
Fresco	11	8
RxJava	8	10
Jenkins	9	9
Spring Sec	8	8
Dubbo	9	10
OkHttp	14	9

Most fell between 8 and 10 refactorings, consistent with the number of elements in our selected context (top-10 degraded elements), suggesting that the recommended effort is adequate.

Finding 4. When considering only the best solutions, NSGA-II and MOSA achieved similar results across all objective functions. However, taking into account the impact on LCOM, computing time, and solution diversity, NSGA-II remains the most effective strategy. Moreover, its results indicate that our refactoring heuristics and mutation operators can effectively recommend refactorings that improve feature modularization and reduce the density of symptoms.

5.4 Qualitative Evaluation of Recommendations

Four collaborators analyzed the best solutions generated by MOSA and NSGA-II for the OkHttp project. In general, feature-driven solutions included refactorings suitable for feature modularization. Extract Class heuristics were often sound and viable, although some recommendations were imprecise.

Best solution of MOSA strategy. The best MOSA solution included 14 refactorings (6 Extract Class and 8 Move Method) across 9 classes: **QT1.** "*Overall, the refactorings can improve feature modularization. However, in some cases some moved methods or extracted classes are not necessary. Some move methods were unrelated to the target class ´ feature. As far as class extractions are concerned, the tool suggested extracting some methods that should stay in the class as they were related to the class' main feature.*"

MOSA's effectiveness was strongly affected by topic model quality. Generic terms sometimes led to incorrect feature associations, resulting in impractical recommendations (e.g., moving *RealCall.cancel()* to *RealWebSocket.CancelRunnable*). This suggests the need for developer inspection of topic models before using *FeatureRef*.

The following quotations illustrate Extract Class recommendations: **QT2.** *"I agree with the extract class from OkHttp's Platform' class. Multiple methods should be placed in other classes, since they deal with features like logging and protocol communication between layers of security. However, the findPlatform' method should stay in the class."* QT2 shows that many recommendations were valid. However, some failed to correctly identify features: **QT3.** *"I disagree with extracting a class from 'Headers'. Most of the methods are related to the building of the headers."*

Best solution of NSGA-II strategy. The best NSGA-II solution consisted of 9 refactorings (8 Extract Class and 1 Move Method), reinforcing the effectiveness of Extract Class heuristics: **QT4.** *"Extract Class on Transmitter' class is recommended because this class implements many responsibilities and the recommended extractions are related to a single responsibility, the Response feature. Thus, it is indicated to extract these fields and methods to another class. Indeed, Transmitter' needs many refactorings to improve its feature modularization because it is a God Class."*

As with MOSA, inaccuracies in feature identification also affected NSGA-II: **QT5.** *"Extract Class on class 'Builder' does not make much sense because these attributes and methods are related to building a response. Thus, it is not appropriate to extracting these fields and methods to another class."* These limitations highlight the need for improved feature-driven heuristics and better topic model configuration, particularly considering locality and syntactic dependencies.

Additional Steps Required. Some recommendations require manual adjustments: **QT6.** *"Compilation errors may happen because some extracted methods use internal methods (from the source class), then the developer need to update the extracted class to call these methods."* This indicates that automated support for applying refactorings could reduce manual effort. Overall, *FeatureRef* produces promising results, and its limitations can be mitigated through improved topic modeling and heuristics, as well as prior techniques for repairing feature mappings [35].

Developers play a central role in the *FeatureRef* workflow, reviewing topic models and validating recommendation side effects, including buildability and test regressions. Our qualitative findings (QT6) indicate that manual adjustments are often needed to resolve dependencies missed by automated heuristics.

Finding 5. The quality of solutions generated by *FeatureRef* depends directly on adequately configuring and training a

topic model for feature identification. However, even in the worst case scenario, *FeatureRef* was able to generate sound and viable recommendations. On the other hand, we need to improve our Move Method and Move Field heuristics, since they presented some undesirable results.

6 Threats to Validity and Limitations

Internal. The configuration of detection algorithms is a common threat in recommendation studies. To mitigate this, we followed settings from prior work [3, 24, 43] and complemented them with trial-and-error tuning. We adopted MOSA and NSGA-II [15, 24] with standard parameters and executed 30 independent runs to reduce convergence bias [4, 15]. Code smell detection rules and thresholds may also affect results; thus, we relied on consolidated rules from the literature, since *FeatureRef* depends on automatically detected symptoms. Also, we plan to use fine-tuned LLMs in the future to improve feature-to-code mappings, albeit a recent study [54] suggests that they are not yet reliable for this task.

Conclusion and External. Automated detection of DP symptoms may introduce false positives and negatives, impacting quantitative results. We mitigated this threat by manually validating a subset of recommendations. We also compare *FeatureRef* against a self-defined baseline, as no existing technique targets feature modularization directly. To reduce this limitation, the baseline was inspired by prior work, and all data and implementation are publicly available (Section 8). We evaluated six open-source projects, which may limit the generalizability of the findings. To mitigate this, we applied a systematic selection process, resulting in a diverse set of Java projects in terms of size, architecture, and domain.

7 Other Related Works

There are several studies on applying refactorings to remove DPs [25, 29, 45, 48, 55]. Kumar and Kumar [25] report an industrial case study on refactoring a payment integration platform, motivated by DPs such as Feature Overload, which resulted in significant improvements in stability and throughput. Lin *et al.* [29] propose an approach that guides developers through step-wise refactorings to transform source code according to a target architectural design. Multiple studies have used search-based algorithms to create refactoring recommendations [1, 3, 24, 43]. Ouni *et al.* [43], for example, define an automated approach for refactoring recommendations, called MORE, using a genetic algorithm with three goals: (1) improve design quality, (2) fix code smells, and (3) introduce design patterns. Alizadeh *et al.* [3] propose a technique called RefBot. It focuses on providing recommendations in the context of pull requests. The technique relies on a software bot that analyzes the pull request changes through a search-based algorithm, which is able to compose refactoring recommendations.

Unlike most refactoring recommendation techniques, which rely on traditional quality measures (e.g., #symptoms, QMOOD, OO metrics) [30], *FeatureRef* combines heterogeneous design symptoms with feature mappings to explicitly support feature modularization. Although some approaches also leverage heterogeneous information [5, 37, 47], they address more limited scenarios, such as recommending only *Move Class* refactorings [5].

The technique proposed by Rebai *et al.* [47] help developers to find and complement incomplete refactorings. They find incomplete refactorings through the developers' refactoring intention manifested in commit messages. The main difference between us is that they focus on completing existing refactoring sequences, while we focus on recommending refactorings for removing DPs. Recent work explores deep learning and transformer-based models for refactoring recommendation, such as ROSE [36], which targets architectural smells. Despite their promising results, these approaches mainly rely on structural or historical data and offer limited interpretability regarding the relationship between refactorings and feature implementation. Consequently, their applicability to feature-related DPs is restricted. Our approach differs by explicitly modeling how features are implemented and distributed across the system, providing targeted feature refactorings.

8 Conclusion

In this paper, we proposed and evaluated *FeatureRef*, a technique for providing effective refactoring recommendations to improve feature modularity. To the best of our knowledge, *FeatureRef* is the first technique that provides feature-driven refactoring recommendations without requiring a complete redesign of the software system. *FeatureRef* incorporates search-based algorithms to explore the vast search space of the refactoring problem, and is flexible enough to allow different strategies for selecting target elements.

The empirical evaluation validated our design decisions, showing that optimized recommendations better improve refactored elements without negatively impacting others. As future work, we envision improving the *Feature Detection* component with state-of-the-art NLP techniques, including additional refactoring types such as Move Class and Extract Method, restricting recommendations to automatically applicable refactorings, and conducting further evaluation studies to improve *FeatureRef* based on empirical evidence. Future work includes integrating the technique into IDEs and CI/CD pipelines to recommend feature-related DPs, report quality warnings, and validate behavioral preservation through automated test execution.

Artifact Availability

The artifacts of this paper include the empirical study data, the tool's source code, and the topic models[5].

Acknowledgments

We would like to sincerely thank William Oizumi for his invaluable theoretical foundation and for laying the groundwork that made this research possible. This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) www.ines.org.br, CNPq grant 408817/2024-0. Authors also are also supported by FAPERJ (200.510/2023, 211.033/2019, 202.621/2019), FUNCAP (BP6-00241-00276.01.00/25), CNPq (201708/2025-6, 403304/2025-3, 404406/2023-8, 310391/2025-3, 404027/2023-7, 306774/2025-9); and CAPES/Proex (88887.373933/2019-00).

[5] https://github.com/opus-research/feature-ref-package

References

[1] Vahid Alizadeh and Marouane Kessentini. 2018. Reducing Interactive Refactoring Effort via Clustering-based Multi-objective Search. In *33rd Int. Conference on Automated Software Engineering*. ACM, 464–474.

[2] V. Alizadeh, M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai. 2018. An Interactive and Dynamic Search-Based Approach to Software Refactoring Recommendations. *IEEE Trans. on Software Engineering* (2018), 932–961.

[3] Vahid Alizadeh, Mohamed Amine Ouali, Marouane Kessentini, and Meriem Chater. 2019. RefBot: intelligent software refactoring bot. In *34th Int. Conference on Automated Software Engineering*. IEEE, 823–834.

[4] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.

[5] Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea de Lucia. 2014. Improving Software Modularization via Automated Analysis of Latent Topics and Dependencies. *ACM Trans. Softw. Eng. Methodol.* (2014).

[6] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2013. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Trans. on Software Engineering* 40, 7 (2013), 671–694.

[7] Ana Carla Bibiano, Wesley K. G. Assunção, Daniel Coutinho, Kleber Santos, Vinícius Soares, Rohit Gheyi, Alessandro Garcia, Baldoino Fonseca, Márcio Ribeiro, Daniel Oliveira, Caio Barbosa, João Lucas Marques, and Anderson Oliveira. 2021. Look Ahead! Revealing Complete Composite Refactorings and their Smelliness Effects. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 298–308. doi:10.1109/ICSME52107.2021.00033

[8] Ana Carla Bibiano, Eduardo Fernandes, Daniel Oliveira, Alessandro Garcia, Marcos Kalinowski, Baldoino Fonseca, Roberto Oliveira, Anderson Oliveira, and Diego Cedrim. 2019. A Quantitative Study on Characteristics and Effect of Batch Refactoring on Code Smells. In *13th Int. Symposium on Empirical Software Engineering and Measurement*. 1–11.

[9] Ana Carla Bibiano, Vinicius Soares, Daniel Coutinho, Eduardo Fernandes, João Lucas Correia, Kleber Santos, Anderson Oliveira, Alessandro Garcia, Rohit Gheyi, Baldoino Fonseca, Márcio Ribeiro, Caio Barbosa, and Daniel Oliveira. 2020. How Does Incomplete Composite Refactoring Affect Internal Quality Attributes?. In *28th Int. Conference on Program Comprehension*. ACM, 149–159.

[10] Ana Carla Bibiano, Anderson Uchôa, Wesley K.G. Assunção, Daniel Tenório, Thelma E. Colanzi, Silvia Regina Vergilio, and Alessandro Garcia. 2023. Composite refactoring: Representations, characteristics and effects on software projects. *Information and Software Technology* 156 (2023), 107134. doi:10.1016/j.infsof.2022.107134

[11] Aline Brito, Andre Hora, and Marco Tulio Valente. 2020. Refactoring Graphs: Assessing Refactoring over Time. In *27th Int. Conference on Software Analysis, Evolution and Reengineering*. 367–377.

[12] D Cedrim. 2018. *Understanding and improving batch refactoring in software systems.* Ph. D. Dissertation. Ph.D. dissertation, PUC-Rio, Brazil.

[13] Lawrence Chung, Brian A Nixon, Eric Yu, and John Mylopoulos. 2012. *Non-functional requirements in software engineering*. Vol. 5. Springer Science & Business Media.

[14] Carlos A Coello Coello, Gary B Lamont, David A Van Veldhuizen, et al. 2007. *Evolutionary algorithms for solving multi-objective problems*. Vol. 5. Springer.

[15] Thelma Elita Colanzi, Wesley K. G. Assunção, Silvia R. Vergilio, Paulo Roberto Farah, and Giovani Guizzo. 2020. The Symposium on Search-Based Software Engineering: Past, Present and Future. *Inf Softw Technol* 127 (2020), 106372.

[16] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolutionary Comp.* 6, 2 (2002), 182–197.

[17] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. 2011. JDeodorant: identification and application of extract class refactorings. In *33rd Int. Conference on Software Engineering*. 1037–1039.

[18] M Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, Boston.

[19] J Garcia, D Popescu, G Edwards, and N Medvidovic. 2009. Identifying Architectural Bad Smells. In *13th European Conference on Software Maintenance and Reengineering*. IEEE.

[20] David E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[21] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and Software Technology* 43, 14 (2001), 833 – 839. doi:10.1016/S0950-5849(01)00189-6

[22] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. 2012. *Search Based Software Engineering: Techniques, Taxonomy, Tutorial.* 1–59.

[23] Hisao Ishibuchi, Noritaka Tsukamoto, and Yusuke Nojima. 2008. Evolutionary many-objective optimization: A short review. In *2008 IEEE congress on evolutionary computation (IEEE world congress on computational intelligence)*. IEEE, 2419–2426.

[24] Marouane Kessentini, Troh Josselin Dea, and Ali Ouni. 2017. A Context-Based Refactoring Recommendation Approach Using Simulated Annealing: Two Industrial Case Studies. In *Genetic and Evolutionary Computation Conference*. ACM, 1303–1310.

[25] M. Raveendra Kumar and R. Hari Kumar. 2011. Architectural Refactoring of a Mission Critical Integration Application: A Case Study. In *4th India Software Engineering Conference*. ACM, 77–83.

[26] M Lanza and R Marinescu. 2006. *Object-Oriented Metrics in Practice.* Springer, Heidelberg.

[27] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *J Syst Softw.* 101 (2015), 193–220.

[28] E. Lim, N. Taksande, and C. Seaman. 2012. A Balancing Act: What Software Practitioners Have to Say about Technical Debt. *IEEE Software* 29, 6 (2012), 22–27.

[29] Yun Lin, Xin Peng, Yuanfang Cai, Danny Dig, Diwen Zheng, and Wenyun Zhao. 2016. Interactive and Guided Architectural Refactoring with Search-Based Recommendation. In *24th Int. Symposium on Foundations of Software Engineering*. ACM, 535–546.

[30] Thainá Mariani and Silvia Regina Vergilio. 2017. A systematic review on search-based refactoring. *Inf Softw Technol* 83 (2017), 14–34.

[31] Marinescu. 2004. Detection strategies: metrics-based rules for detecting design flaws. In *20th Int. Conference on Software Maintenance*. 350–359.

[32] Andrew Kachites McCallum. 2002. Mallet: A machine learning for language toolkit (2002).

[33] Kaisa Miettinen, Francisco Ruiz, and Andrzej P Wierzbicki. 2008. Introduction to multiobjective optimization: interactive approaches. In *Multiobjective optimization: interactive and evolutionary approaches*. Springer, 27–57.

[34] E. Murphy-Hill, C. Parnin, and A. P. Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (Jan 2012), 5–18. doi:10.1109/TSE.2011.41

[35] Camila Nunes, Alessandro Garcia, Carlos Lucena, and Jaejoon Lee. 2014. Heuristic expansion of feature mappings in evolving program families. *Software: Practice and Experience* 44, 11 (2014), 1315–1349.

[36] Samal Nursapa, Anastassiya Samuilova, Alessio Bucaioni, and Phuong T Nguyen. 2025. ROSE: Transformer-Based Refactoring Recommendation for Architectural Smells. In *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 421–427.

[37] Ally S Nyamawe, Hui Liu, Nan Niu, Qasim Umer, and Zhendong Niu. 2019. Automated recommendation of software refactorings based on feature requests. In *27th Int. Requirements Engineering Conference*. IEEE, 187–198.

[38] Willian Oizumi, Diego Cedrim, Leonardo Sousa, Ana Carla Bibiano, Anderson Oliveira, Alessandro Garcia, and Daniel Tenorio. 2020. Recommending Composite Refactorings for Smell Removal: Heuristics and Evaluation. In *34th Brazilian Symposium on Software Engineering (SBES)*.

[39] W. Oizumi, A. Garcia, L. S. Sousa, B. Cafeo, and Yixue Zhao. 2016. Code Anomalies Flock Together: Exploring Code Anomaly Agglomerations for Locating Design Problems. In *38th Int. Conference on Software Engineering*.

[40] Willian Oizumi, Leonardo Sousa, Anderson Oliveira, Luiz Carvalho, Alessandro Garcia, Thelma Colanzi, and Roberto Oliveira. 2019. On the density and diversity of degradation symptoms in refactored classes: A multi-case study. In *30th Int. Symposium on Software Reliability Engineering*.

[41] Willian Oizumi, Leonardo Sousa, Anderson Oliveira, Alessandro Garcia, Anne Benedicte Agbachi, Roberto Oliveira, and Carlos Lucena. 2018. On the identification of design problems in stinky code: experiences and tool support. *Journal of the Brazilian Computer Society* 24, 1 (22 Oct 2018), 13. doi:10.1186/s13173-018-0078-y

[42] Anderson Oliveira, Willian Oizumi, Leonardo Sousa, Wesley KG Assunção, Alessandro Garcia, Carlos Lucena, and Diego Cedrim. 2022. Smell Patterns as Indicators of Design Degradation: Do Developers Agree?. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*. 311–320.

[43] Ali Ouni, Marouane Kessentini, Mel Ó Cinnéide, Houari Sahraoui, Kalyanmoy Deb, and Katsuro Inoue. 2017. MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. *Journal of Software: Evolution and Process* 29, 5 (2017), e1843.

[44] D. L. Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. doi:10.1145/361598.361623

[45] P. Rachow. 2019. Refactoring Decision Support for Developers and Architects Based on Architectural Impact. In *Int. Conference on Software Architecture Companion*. 262–266.

[46] Narayan Ramasubbu, Marcelo Cataldo, Rajesh Krishna Balan, and James D. Herbsleb. 2011. Configuring global software teams: a multi-company analysis of project productivity, quality, and profits. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 261–270. doi:10.1145/1985793.1985830

[47] Soumaya Rebai, Marouane Kessentini, Vahid Alizadeh, Oussama Ben Sghaier, and Rick Kazman. 2020. Recommending refactorings via commit message analysis. *Information and Software Technology* 126 (2020), 106332.

[48] Luca Rizzi, Francesca Arcelli Fontana, and Riccardo Roveda. 2018. Support for Architectural Smell Refactoring. In *Proc. of the 2nd Int. Workshop on Refactoring*

(IWoR 2018). ACM, 7–10.

[49] Camila Costa Silva, Matthias Galster, and Fabian Gilson. 2021. Topic modeling in software engineering research. *Empir. Softw. Eng.* 26, 6 (2021), 1–62.

[50] Leonardo Sousa, Diego Cedrim, Alessandro Garcia, Willian Oizumi, Ana Carla Bibiano, Daniel Tenorio, Miryung Kim, and Anderson Oliveira. 2020. Characterizing and Identifying Composite Refactorings: Concepts, Heuristics and Patterns. In *17th Int. Conference on Mining Software Repositories*.

[51] Leonardo Sousa, Anderson Oliveira, Willian Oizumi, Simone Barbosa, Alessandro Garcia, Jaejoon Lee, Marcos Kalinowski, Rafael de Mello, Baldoino Fonseca, Roberto Oliveira, Carlos Lucena, and Rodrigo Paes. 2018. Identifying Design Problems in the Source Code: A Grounded Theory. In *40th Int. Conference on Software Engineering*. ACM, 921–931.

[52] Leonardo Sousa, Roberto Oliveira, Alessandro Garcia, Jaejoon Lee, Tayana Conte, Willian Oizumi, Rafael de Mello, Adriana Lopes, Natasha Valentim, Edson Oliveira, and Carlos Lucena. 2017. How Do Software Developers Identify Design Problems?: A Qualitative Analysis. In *31st Brazilian Symposium on Software Engineering*.

[53] Ekunda Lukata Ulungu, JFPH Teghem, PH Fortemps, and Daniel Tuyttens. 1999. MOSA method: a tool for solving multiobjective combinatorial optimization problems. *Journal of multicriteria decision analysis* 8, 4 (1999), 221.

[54] Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, et al. 2025. Towards Understanding the Characteristics of Code Generation Errors Made by Large Language Models. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE 2025)*. IEEE/ACM, 2587–2599. doi:10.1109/ICSE55347.2025.00180

[55] Olaf Zimmermann. 2017. Architectural refactoring for the cloud: a decision-centric view on cloud migration. *Computing* 99, 2 (01 Feb 2017), 129–145.