Mining and Managing Big Data Refactoring for Design Improvement: Are We There Yet?

Eman Abdullah AlOmar[1], Mohamed Wiem Mkaouer[1], and Ali Ouni[2]

[1] Rochester Institute of Technology,
Rochester, New York, USA
[2] ETS Montreal, University of Quebec,
Montreal, QC, Canada

Abstract. Refactoring is a set of code changes applied to improve the internal structure of a program, without altering its external behavior. With the rise of continuous integration and the awareness of the necessity of managing technical debt, refactoring has become even more popular in recent software builds. Recent studies indicate that developers often perform refactorings. If we consider all refactorings performed across all projects, this consists of the refactoring knowledge that represents a rich source of information that can be useful for both developers and practitioners to better understand how refactoring is being applied in practice. However, mining, processing, and extracting useful insights, from this plethora of refactorings, seems to be challenging. In this book chapter, we take a dive into how refactoring can be mined and preprocessed. We discuss all design concepts and structural metrics that can also be mined along with refactoring operations to understand their impact better. We further investigate the many practical challenges for such extraction. The volume, velocity, and variety of extracted data require careful planning. We outline the appropriate techniques from a large number of available technologies for such system implementation.

Keywords: refactoring, software maintenance, software quality

1 Introduction

Successful software systems undergo evolution through the continuous code changes, as means to update features, fix bugs, and produce a more reliable and efficient product. Prior studies have pointed out how software complexity can be a serious obstacle preventing the ease of software evolution, as large and sophisticated modules are, in general, harder to understand, and error-prone. Such patterns, located in the system design, negatively impact the overall quality of software as they are responsible for making its design inadequate for evolution. In this context, it has been shown that software engineers spend up to 60% of their programming time in reading source code, and trying to understand its functionality, in order to properly perform the needed changes without "breaking" the code. Consequently, software maintenance activities that are related to improving the overall software quality to take up to 67% of the cost allocated for the project. The *de-facto* way of handling such debt is through software refactoring. By definition, refactoring is the art of improving design structure while preserving the overall external behavior. With the rise of technical debt, and developers acknowledgment of shortage in their deliverables, refactoring stands as a critical task to maintain the existence of software and to prevent it from decay.

Projects that are known to be successful in maintaining their quality through several waves of updates and migrations across various programming paradigms and frameworks, are known to be witnessing efficient refactoring strategies. Such hidden knowledge has triggered the intention of research to mine and understand how developers refactor their code in practice. In this context, several refactoring detection tools have been lately proposed to mine the development history of a given software project, and extract all the information related to all refactoring operations that were performed on its code elements.

As recent refactoring tools (e.g., RefactoringMiner [24] and RefDiff [21]) have reached a high level of maturity, their usage across various large projects has triggered an explosion in the information that can be obtained regarding previously performed refactorings, and their corresponding impact on the source code. Furthermore, refactoring, being by nature a code change, when batched, becomes harder to analyze. Moreover, code changes visualization is gaining more attention in software engineering research, yet visualizing refactoring is still under-researched.

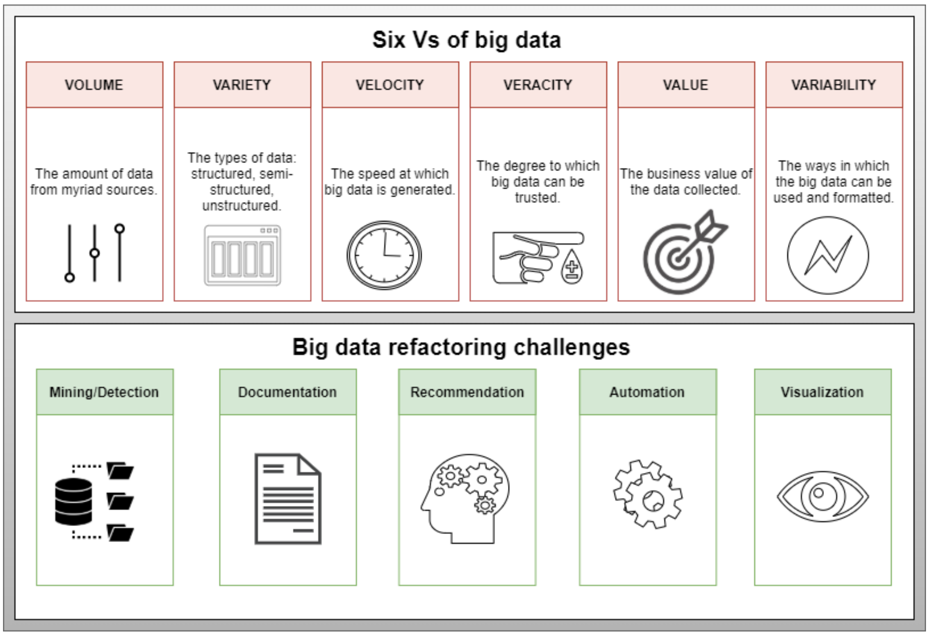
Six Vs of big data
VOLUME
The amount of data from myriad sources.
VARIETY
The types of data: structured, semi-structured, unstructured.
VELOCITY
The speed at which big data is generated.
VERACITY
The degree to which big data can be trusted.
VALUE
The business value of the data collected.
VARIABILITY
The ways in which the big data can be used and formatted.
Big data refactoring challenges
Mining/Detection
Documentation
Recommendation
Automation
Visualization

Fig. 1: Big data refactoring

For the above mentioned challenges that the plethora of refactorings have emerged, this chapter initiates the discussion about how the world of big data can provide a rich source of solutions. We detail the multiple challenges linked to refactoring indexing, analysis and visualization, while exploring potential big data solutions. As depicted in Figure 1, we identify five refactoring challenges, triggering the explosion of refactoring data, which we can call *Big Data Refactoring Challenges*. These challenges are 1) Detection of refactoring operations in software systems, 2) Developer's Documentation of refactoring activities, 3) Recommendation of refactoring opportunities on existing software systems, 5) Automation of refactoring execution, and 5) Visualization of refactoring impact on the source code. We organize this chapter to explore each of these challenges, by detailing its existing tools and methodologies, along with discussing their limitations and how they are explicitly or implicitly linked to big data dimensions.

This chapter is organized as follows: the first section is associated with tools and techniques related to the identification of executed refactorings, the next section is dedicated for documentation. Section 4 summarizes the existing tools to automate the generation of refactorings. Recommending refactorings is also covered in Section 5. The need for refactoring visualization is covered in Section 6 before concluding in Section 7.

2 Mining and Detection

The popularity of the GitHub hosting service is increasing rapidly and has been used frequently for the base of data collection in literature. Research in mining software repositories mainly relies on two GitHub services: the version and bug tracking systems. GitHub stores all versions of the source code and any specific changes are represented by a commit that involves a textual description of the change (i.e., commit message). The bug tracking system, on the other hand, provides an interface for reporting errors. GitHub makes it possible to mine a large amount of information and different properties of open source projects.

The challenge in this area lies in analyzing a comprehensive and large number of GitHub commits containing refactoring. Several studies have mining tools to identify refactoring operations between two versions of a software system. Dig et al. [9] developed a tool called Refactoring Crawler, which uses syntax and graph analysis to detect refactorings. Prete et al. [20] proposed Ref-Finder, which identifies complex refactorings using a template-based approach. Hayashi et al. [12] considered the detection of refactorings as a search problem. The authors proposed a graph search algorithm to model changes between software versions. Xing and Stroulia [27] proposed JDevAn, which is a UMLDiff based, design-level analyzer for detecting refactorings in the history of Object-Oriented systems. Tsantalis et al. presented RefactoringMiner, which is a lightweight, UMLDiff based algorithm that mines refactorings within Git commits. Silva and Valente [21] extended RefactoringMiner by combining the heuristics-based static analysis with code similarity (TF-IDF weighting scheme) to identify 13 refactoring types. Tsantalis et al. [24] extended their tool to enhance the accuracy of the 28 refactoring types that can be detected through structural constraints. A recent survey by Tan [22] compares several refactoring detection tools and shows that RefactoringMiner is currently the most accurate refactoring detection tool. The choice of the mining tool is driven by accuracy; therefore RefactoringMiner is suitable for mining and detecting refactorings and extracting big data refactoring. It is suitable for studies that require a large variety of repositories and commit volumes.

Table 1: Studied dataset statistics.

Item	**Count**
Studied projects	3,795
Commits with refactorings	322,479
Refactoring operations	1,208,970
Commits with refactorings & Keywords	2,312
Remove false positive commits	1,067
Final dataset	1,245

With the existence of millions of software projects, whose sizes vary from small to large, mining their refactorings could lead to an amount of data that cannot be handled by traditional means. This links mining refactoring to Big Data's Volume. For instance, in our recent study [2], we mined refactoring in 3,795 open source projects. The process extracted over 1,200,000 refactoring operations, distributed in 322,479 commits. More details about this dataset is in Table 1. We faced challenges in hosting and querying this data. To extend our study, we need to extract refactorings in over 300 000 open source projects, and we are currently unable to perform this study, unless we seek the right framework to collect, store, and index such data.

Another interesting challenge related to such data, is its heterogeneity. Refactoring operations are different from each other in their structure, target code elements and impact on source code. For instance, the rename identifier refactoring, is the act of changing the name of a given attribute. Such operation requires saving the old name of the attribute, its new name and the path of the file containing the attribute. As for extract method, which is the splitting of a given method into two sub-methods, this operation requires saving the old method signature, and body (and path) along with saving the signature and bodies of the newly created methods (and paths). So, each refactoring type requires a unique structure to store its information. Furthermore, various studies are interested in the reachability of the refactoring operation, to better analyze their impact on the code design. Storing refactored code elements and their corresponding dependencies may require specific data structures like graphs. For large and complex systems, analyzing such information is challenging.

3 Refactoring Documentation

A number of studies have focused recently on the identification and detection of refactoring activities during the software life cycle. One of the common approaches to identify refactoring activities is to analyze the commit messages in version-controlled repositories. Prior work [2] has explored how developers document their refactoring activities in commit messages; this activity is called Self-Admitted Refactoring or Self-Affirmed Refactoring (SAR). In particular, SAR refers to the situation which shows developers explicit documentation of refactoring operations intentionally introduced during a code change. For example, by manual inspection of the Cassandra-unit1 open source project, AlOmar et al [2] used this example to demonstrate SAR: "refactoring of Abstract*DataSet to delete duplicate code," which indicates that developers intentionally refactor one class to remove the redundancy antipattern that violates design principles. The authors manually analyzed commit messages by reading through 58,131 commits. Then they extracted, from these commit messages, a set of repetitive keywords and phrases which are specific to refactoring. They provided a set of 87 patterns, identified across 3,795 open source projects. Since this approach heavily depends on the manual inspection of commit messages, in follow-up work, AlOmar et al. [3] presented a two-step approach that firstly distinguishes whether a commit message potentially contains an explicit description of a refactoring effort. Then, secondly classifies it into one of the three common categories identified in previous study [2], which is the first attempt to automate the detection and classification of self-affirmed refactorings. The existence of such patterns unlocks more studies that question the developers' perception of quality attributes (e.g., coupling, complexity); these results may be used to recommend future refactoring activity. For instance, AlOmar et al. [4] identified which quality models are more in line with the developer's vision of quality optimization when they explicitly mention in the commit messages that they refactor to improve these quality attributes. This study shows that, although there is a variety of structural metrics can represent internal quality attributes, not all of them can measure what developers consider to be an improvement in their source code. Based on their empirical investigation, for metrics that are associated with quality attributes, there are different degrees of improvement and degradation of software quality for different SAR patterns.

As stated above, developers use a variety of patterns to express their refactoring activities. Previous studies illustrate such a pattern. However, one big challenge is that it is not practical for large real world projects to manually collect all potential keywords/phrases reported in a large number of commit messages, as developers may use various expressions to annotate how they refactor. To cope with this challenge, future research could plan to use the findings of previous studies to build a text-mining tool that will automatically support software engineers in the task of identifying, detecting, and highlighting self-affirmed refactoring in the commit messages. This detector could allow users to train their own model and integrate self-affirmed refactoring detectors into their development tools.

Table 2: Potential candidate refactoring text patterns.

Potential Candidate Refactoring Patterns

BugFix	**Code Smell**	**External**	**Functional**	**Internal**
Minor fixes	**Avoid code duplication**	Reusable structure	Add* feature	**Decoupling**
Bug* fix*	**Avoid duplicate code**	Improv* code reuse	Add new feature	Enhance loose coupling
Fix* bug*	Avoid redundant method	Add* flexibility	Added a bunch of features	Reduced coupling
Bug hunting	**Code duplication removed**	Increased flexibility	**New module**	Reduce coupling and scope of responsibility
Correction of bug	**Delet* duplicate code**	**More flexibility**	**Fix some GUI**	Prevent the tight coupling
Improv* error handling	Remove unnecessary else blocks	Provide flexibility	Added interesting feature	Reduced the code size
Fix further thread safety issues	**Eliminate duplicate code**	A bit more readable	Added more features	Complexity has been reduced
Fixed major bug	Fix for duplicate method	**Better readability**	Adding features to support	**Reduce complexity**
Fix numerous bug	Filter duplicate	Better readability and testability	Adding new features	Reduced greatly the complexity
Fix several bug	Joining duplicate code	Code readability optimization	Addition of a new feature	Removed unneeded complexity
Fixed a minor bug	Reduce a ton of code duplication	Easier readability	Feature added	Removes much of the complexity
Fixed a tricky bug	**Reduce code duplication**	**Improve readability**	Implement one of the batch features	Add inheritance
Fix* small bug	Reduced code repetition	**Increase readability**	Implementation of feature	Added support to the inheritance
Fixed nasty bug	**Refactored duplicate code**	Make it better readable	Implemented the experimental feature	Avoid using inheritance and using composition instead
Fix* some bug*	Clear up a small design flaw	Make it more readable	Introduced possibility to erase features	Better support for specification inheritance
Fixed some minor bugs	TemporalField has been refactored	Readability enhancement	New feature	**Change* inheritance**
bugfix*	Remove commented out code	Readability and supportability improvement	Remove the default feature	Extend the generated classes using inheritance
Fix* typo*	Removed a lot of code duplication	**Readability improvements**	Removed incomplete features	Improved support for inheritance
Fix* broken	**Remov* code duplication**	Reformatted for readability	Renamed many features	Perform deep inheritance
Fix* incorrect	Remove some code duplication	Simplify readability	Renamed some of the features for consistency	Remove inheritance
Fix* issue*	Removed the big code duplication	Improve* testability	Small feature addition	Inheritance management
Fix* several issue*	Removed some dead and duplicate code	Update the performance	Support of optional feature	Loosened the module dependency
Fix* concurrency issue* with	**Remov* duplicate code**	Add* performance	Supporting for derived features	Prevents circular inheritance
Fixes several issues	Resolved duplicate code	Scalability improvement	Added functionality	Avoid using inheritance
Solved some minor bugs	Sort out horrendous code duplication	Better performance	Added functionality for merge	Add composition
Working on a bug	Remove duplicated field	Huge performance improvement	Adding new functionality	Composition better than inheritance
Get rid of	Remov* dead code	**Improv* performance**	Adds two new pieces of functionality to	Us* composition
A bit of a simple solution to the issue	Remove some dead-code	**More manageable**	Consolidate common functionality	Separates concerns
A fix to the issue	Removed all dead code	**More efficient***	Development of this functionality	Better handling of polymorphism
Fix a couple of issue	Removed apparently dead code	Make it reusable for other	Export functionality	Makes polymorphism easier
Issue management	This is a bit of dead code	Increase efficiency	Extend functionality of	**Better encapsulation**
Fix* minor issue	Removed more dead code	Verify correctness	Extract common functionality	Better encapsulation and less dependencies
Correct issue	Fix* code smell	Massive performance improvement	Functionality added	Pushed down dependencies
Additional fixes	Fix* some code smell	Increase performance	House common functionality	**Remov* dependency**
Resolv* problem	Remov* some 'code smells'	Largely resolved performance issues	Improved functionality	Split out each module into
Correct* test failure*	Update data classes	Lots of performance improvement	**Move functionality**	
Fix* all failed test*	Remove useless class	Measuring the performance	Moved shared functionality	
Fix* compile failure	Removed obviously unused/-faulty code	Improv* stability	Feature/code improvements	
A fix for the errors	Lots of modifications to code style	Usability improvements	Pulling-up common functionality	
Better error handling	Antipattern bad for performances	Noticeable performance improvement	Push more functionality	
Better error message handling	Killed really old comments	Optimizing the speed	Re-implemented missing functions	
Cleanup error message	Less long methods	Performance boost	Refactored functionality	
Error fix*	Removed some unnecessary fields	**Performance enhancement**	Refactoring existing functionality	
Fixed wrong		Performance improvement	Add functionality to	
Fix* error*		Performance much improved	Remov* function*	
Fix* some error*		Performance optimization	Merging its functionality with	
Fix small error		Performance speed-up	Remove* unnecessary function*	
Fix some errors		Refactor performance test	Reworked functionality	
Fix compile error		Renamed performance test	Removing obsolete functionality	
Fix test error		Speed up performance	Replicating existing functionality with	
Fixed more compilation errors		Backward compatible with	Split out the GUI function	
Fixed some compile errors		Fix backward compatibility	Add cosmetic changes	
Fixes all build errors		Fixing migration compatibility	Add* support	
Fixed Failing tests		Fully compatible with	Implement* new architecture	
Handle		Keep backwards compatible	Update	
Handling error*		Maintain compatibility	Additional changes for	
Error* fix*		Make it compatible with	UI layout has changed	
Tweaking error handling		**More compatible**	GUI: Small changes	
Various fix*		Should be backward-compatible	New UI layout	
Fix* problem*		Retains backward compatibility	**UI changes**	
Got rid of deprecated code		Stay compatible with	UI enhancements	
Delet* deprecated code		Added some robustness		
Remov* deprecated code		Improve robustness		
		Improve usability		

If we want to extend the study of AlOmar et al. [4], and analyze refactoring documentation across the dataset previously described in Table 1, we are challenged by the Volume of text that needs to be analyzed. Furthermore, this text is originated from many developers, from different projects, and so, it contains various semantics, which increases the ambiguity of deciphering it. From a Variability perspective, there is a need to find better formatting and indexing for this text in order to adequately extract the needed information. For instance, the rise of word2Vec [11], when combined with the appropriate vector indexing, may provide a potential solution to avoid naive string matching, which is known to generate false positives. Other topic modeling techniques can be also explored to extract textual patterns which are relevant to refactoring documentation, however, their manual validation is challenging due to the large number of potential patterns that can be generated. For instance, Table 2, showcases the existence of various refactoring candidate textual patterns, extracted from our dataset in Table 1, and which require manual validation.

4 Refactoring Automation

Maintaining large scale code and ensuring large scale semantically safe refactoring can be a challenging task. Many contemporary IDEs provide a limited set of automatic refactoring operations applied to a single file or package. Handling large refactoring poses a big challenge in many object-oriented development projects. Further, performing a high volume of refactoring typically takes longer and changes multiple parts of the system. If refactoring influences large chunks of the system, as a result, there is a need to break changes down into smaller parts. A few questions could be investigated when performing Volume and Variety of refactoring:

– How can large refactoring operations be planned?
– How can undo-functionality be implemented for large refactorings during the actual refactoring?
– How can we proceed to add more functionality during the execution of large refactorings while ensuring behavior preservation for the existing application?
– How can we integrate the plans of implementing large refactorings into the development process?
– How can we document the status of a large refactoring?

4.1 Refactoring Tools

Various aspects of refactoring need to be considered when automating the application of refactoring. These include, but are not limited to, automation, reliability, coverage, and scalability of refactoring tools. With regards to automation, fully automated and semi-automated refactoring tools are beneficial for developers. For example, adding support of an "undo" feature can facilitate the process of returning the software to its original state in case the effect of refactoring is not desirable. Reliability indicates whether the software guarantees behavior preservation of the refactoring transformation. A full guarantee of behavioral preservation is challenging, thus, an automated refactoring tool should define a set of pre and post conditions to ensure program correctness after the application of refactoring. Concerning coverage, refactoring tools should cover a wide range of refactoring activities that developers could perform, i.e., the tool should be as complete as possible. It would be worthwhile to have refactoring tools that support a complete set of refactoring operations of different levels of granularity (e.g., class, method, package) to improve the system design from different perspectives (e.g., code smell removal, adherence to object-oriented design practices such as SOLID and GRASP, etc). Scalability is another aspect that should be taken into consideration when constructing refactoring tools.

4.2 Lack of Use

Despite the positive aspects of semi-automated refactoring, many developers continue to prefer to do refactoring manually, even when the opportunity to use a refactoring tool presents itself. In the realm of Extract Method refactoring, Kim et al. [13] found that 58.3% of developers chose to perform their refactorings manually. Another study by Negara et al. [18] shows that even though the majority of developers aware of refactoring tools and their benefits, they still chose to refactor manually. Murphy-Hill et al. [16] found that only 2 out of 16 students in an object-oriented programming class had previously used refactoring tools. Another survey by Murphy-Hill [15] found that 63% of surveyed individuals at an Agile methodology conference used environments with refactoring tools, and that they use the tools 68% of the time when one is available. This is significant, since Agile methodologies are generally predisposed to be in favor of refactoring, indicating the general usage must be even lower. Murphy-Hill tempers this statement by noting the likelihood of bias in the participants' responses, as well as the survey size of 112 being non-representative as it is comparatively small compared to all programmers.

Murphy-Hill also compared studies by Murphy-Hill et al. and Mäntylä et al. They show that students claim they are more likely to perform Extract Method refactoring immediately compared to Rename refactoring, yet developers are 8 times as likely to use a Rename refactoring tool than an Extract Method refactoring tool [14]. Research by Vakilian et al. and Kim et al. also indicate that the majority of developers would prefer to apply refactorings other than Rename refactoring manually [25, 13]. There is no clear conclusion for this discrepancy, but it indicates either an underuse of Extract Method refactoring tools or overuse of Rename refactoring tools. Ultimately, it seems unrealistic to come to a concrete conclusion regarding the use of refactoring tools by all developers, but these findings show strong indirect evidence that refactoring tools are underutilized compared to their potential.

From big data perspective, these studies suffer from lack of analysis of Value. There should be an alignment of how tools refactor code with what developers are expecting their code to be refactored. So far, existing tools focus on removing code smells, and improving the design structural measurements, however, and as seen in Table 2, developers do refactor their code for various reasons that go beyond these two objectives.

4.3 Lack of Trust

There have been a number of studies and surveys done collecting information on developers' aversion to refactoring tools. Surveys by Campbell et al. [8], Pinto et al. [19], and Murphy-Hill [15] include the same barrier to entry in their findings: lack of trust . In general, this refers to when a developer is unwilling to give control over modification of the code base to the refactoring tool due to perceived potential problems. This can manifest for a number of reasons. The developer may be unfamiliar with the tool and unwilling to risk experimenting with a tool that could modify the program in unexpected ways. The developer may be unfamiliar with the terms the tool uses, or the information it displays, or the tool may be difficult to learn or use. They may not understand exactly what the tool intends to change about their program. They may not know how the tool will affect the style or readability of the code, or they may be familiar with this and knowingly dislike what it will do to their code. Pinto et al. [19] found that some developers will avoid suggested refactorings if they would need to trade readability for atomicity. In any of these scenarios, a more trustworthy option for the developer would be to rely on their own intuition, abilities, and experience.

Developers also reported concerns that refactoring tools would implement poor design choices, either due to bugs in the tool, inconsistencies with the detection algorithms, or special cases with the code base, such as reflection. Several popular refactoring tools have been shown to contain such bugs that modify program behavior without the developer ever knowing [26, 1]. Veracity, or the extent to which refactorings can be trusted, is emerging problem from big data perspective.

4.4 Behaviour Preservation

Today, a wide variety of refactoring tools automates several aspects of refactoring. However, ensuring the behavior preserving property when building tool-assisted refactoring is challenging. Several formalisms and techniques have been proposed in the existing literature to guarantee the behavior preservation and correctness of refactorings. Actual source code transformation and a set of preconditions are the two main parts for any refactoring operation to be performed by automated refactorings.

5 Refactoring Recommendation

Performing refactoring in a large software system can be very challenging. While refactoring is being applied by various developers [5], it would be interesting to evaluate their refactoring practices. We would like to capture and better understand the code refactoring best practices and learn from these developers so that we can recommend them for other developers. There is a need to build a refactoring recommendation system to (1) identify refactoring opportunities and pinpoint design flaws, and (2) apply refactoring solutions. To support future refactorings, structural, semantic, dynamic, and historical information between code components need to be considered. Recent proposed recommenders do generate a large list of refactorings to apply. This represents a challenge for practitioners, since they do not want to lose the identity of their design, also they cannot fully understand the impact of such large set of code changes. Such a problem is mapped to big data Volume and Veracity. Furthermore, running such set of refactoring, requires handling several constraints. it is to satisfy the correctness of the applied refactorings. Previous studies distinguish between two kinds of constraints: structural constraints and semantic constraints. Structural constraints were extensively investigated in the literature. Fowler, for example, defined in [10] a set of pre and post-conditions for a large list of refactoring operations to ensure the structural consistency. However, software engineers should check manually all actors related to the refactoring operation to inspect the semantic relationship between them. In the next subsections, we further detail the challenges of establishing the relationship between refactoring and its corresponding target code element(s).

5.1 Structural Relationship

Structural relationships mean selecting quality metrics to measure system improvement before and after the application of refactoring that includes method calls, shared instance variables, or inheritance relationships. Several quality metrics have been reported in the literature to capture different aspects of internal quality attributes. For example, the coupling between object (CBO) metric correlates with coupling, i.e., the higher the CBO value, the higher the coupling between classes.

5.2 Semantic Relationship

To determine the semantic relationship between code components, textual similarity is measured. If the terms of two code components (i.e., class or method) are very similar, then it is probable that developers used the same terms to express the responsibilities implemented by the class or the method. For example, two methods are considered conceptually related if both of these methods perform conceptually similar actions. This information is useful for grouping similar code components together. There are a few quality metrics to capture semantic similarity (e.g., the conceptual cohesion of classes (C3) and the conceptual coupling between classes (CCBC)). For example, in order to recommend Move Class refactoring, software module classes having high CCBC values can be grouped together. Consequently, the changes can be localized easily by developers and the software will be more manageable and maintainable.

5.3 Historical Information

The refactoring process can be automated, not only by using the state-of-the-art features (improving design metrics and quality attributes) but also with contextual features that simulate developers' presence by using refactoring operations previously performed by developers. These refactoring operations could be obtained by using refactoring-mining tools such as RefactoringMiner and RefDiff that identify refactoring applied between two subsequent versions of a software system.

6 Refactoring Visualization

Visualizing refactoring activity applied to the source code helps provide a big picture about refactoring. It helps gain insight about the source code and improves the understandability of the software. However, visualizing large refactoring activity presents both technical and cognitive challenges. Particularly, if the code change is complex and large, the task of detecting refactoring anomalies and looking for defects becomes more challenging. Developers could perform batch refactoring or sequence of refactoring operations. Murphy-Hill et al. [17] define batch refactorings as refactoring operations that are executed within 60 seconds of each other. Their findings show that developers repeat the application of refactoring, and 40% of refactorings performed using a refactoring tool occur in batches. Recently, Brito et al. [7] introduced a refactoring graph concept to assess refactoring over time. The authors analyzed 10 Java projects, extracted 1,150 refactoring subgraphs, and evaluated their properties: size, commits, age, composition, and developers. To increase the trust between developers and the tool, Bogart et al. [6] recently extended JDeodorant tool by providing developers with the possibility of verifying their refactoring outcomes. The extended tools provide timely visualization of multiple selected refactorings, and detects whether there is a conflict or not.

Visualizing big data refactoring is not deeply studied or discussed in the refactoring literature. Refactoring visualization is a vital process since it allows developers to look at the code and learn how it is organized and how it works. Further, it assists developers in pinpointing possible bad code smells that violate design principles, determining which code paths are susceptible to a bug, and saving development time.

Research in refactoring should expand on refactoring graphs at the method level, and focus on class and package level refactorings. Also, research could complement existing git-based (e.g., RefactoringMiner [24] and RefDiff [21]) and contemporary IDE refactoring tools (e.g., JDeodorant [23] and RefFinder [20]) with visualization features.

7 Conclusion

In this chapter, we have explored various challenges that the rise of refactoring research has been facing, and which represent interesting research opportunities for the big data community. For each refactoring challenge, we explored its related studies to understand its growth and complexity, then we discussed how it is linked to big data dimensions. As we established stronger connections between refactoring and big data, we hope to see emerging studies leveraging big data techniques and frameworks to take refactoring research to the next level.

Bibliography

[1] Rafi Almhana, Wiem Mkaouer, Marouane Kessentini, and Ali Ouni. Recommending relevant classes for bug reports using multi-objective search. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 286–295. IEEE, 2016.

[2] Eman Abdullah Alomar, Mohamed Wiem Mkaouer, and Ali Ouni. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *Proceedings of the 3rd International Workshop on Refactoring*, New York, NY, USA, 2019. ACM.

[3] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software*, page 110821, 2020.

[4] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Maroune Kessentini. On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.

[5] Eman Abdullah AlOmar, Anthony Peruma, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. On the relationship between developer experience and refactoring: An exploratory study and preliminary results. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 342–349, 2020.

[6] Alex Bogart, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Increasing the trust in refactoring through visualization. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 334–341, 2020.

[7] Aline Brito, Andre Hora, and Marco Tulio Valente. Refactoring graphs: Assessing refactoring over time. *arXiv preprint arXiv:2003.04666*, 2020.

[8] Dustin Campbell and Mark Miller. Designing refactoring tools for developers. In *Proceedings of the 2nd Workshop on Refactoring Tools*, page 9. ACM, 2008.

[9] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. *Automated Detection of Refactorings in Evolving Components*, pages 404–428. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[10] Martin Fowler, Kent Beck, John Brant, William Opdyke, and don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[11] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[12] Shinpei Hayashi, Yasuyuki Tsuda, and Motoshi Saeki. Search-based refactoring detection from source code revisions. *IEICE TRANSACTIONS on Information and Systems*, 93(4):754–762, 2010.

[13] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 50. ACM, 2012.

[14] Mika V Mäntylä and Casper Lassenius. Drivers for software refactoring decisions. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 297–306. ACM, 2006.

[15] Emerson Murphy-Hill. Programmer friendly refactoring tools. 2009.

[16] Emerson Murphy-Hill and Andrew P Black. Refactoring tools: Fitness for purpose. *IEEE software*, 25(5), 2008.

[17] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2011.

[18] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E Johnson, and Danny Dig. A comparative study of manual and automated refactorings. In *European Conference on Object-Oriented Programming*, pages 552–576. Springer, 2013.

[19] Gustavo H Pinto and Fernando Kamei. What programmers say about refactoring tools?: An empirical investigation of stack overflow. In *Proceedings of the 2013 ACM workshop on Workshop on refactoring tools*, pages 33–36. ACM, 2013.

[20] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, Sept 2010.

[21] D. Silva and M. T. Valente. Refdiff: Detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 269–279, May 2017.

[22] Liang Tan and Christoph Bockisch. A survey of refactoring detection tools. In *Software Engineering*, 2019.

[23] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *2008 12th European Conference on Software Maintenance and Reengineering*, pages 329–331. IEEE, 2008.

[24] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, pages 483–494. ACM, 2018.

[25] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P Bailey, and Ralph E Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, pages 233–243. IEEE Press, 2012.

[26] Mathieu Verbaere, Ran Ettinger, and Oege De Moor. Jungl: a scripting language for refactoring. In *Proceedings of the 28th international conference on Software engineering*, pages 172–181. ACM, 2006.

[27] Zhenchang Xing and Eleni Stroulia. Umldiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 54–65, New York, NY, USA, 2005. ACM.