Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

European Master in Software Engineering

Master Thesis

Enhancing Code Quality through Automated Refactoring Techniques

Author: Ksenia Tsybulka

June, 2024

This Master Thesis has been deposited in ETSI Informáticos de la Universidad Politécnica de Madrid.

Master Thesis

European Master in Software Engineering

Title:

Enhancing Code Quality through Automated Refactoring Techniques

June / 2024

Author: Ksenia Tsybulka

Supervisor:
Angelica de Antonio Jimenez

Academic title:
Associate Professor

University of the presented title:
Universidad Politécnica de Madrid

Department / School:
Lenguajes y Sistemas Informáticos
e Ingeniería de Software

University:
Universidad Politécnica de Madrid

Abstract

In the context of the rapidly evolving field of software development, maintaining high code quality is of critical importance in order to ensure the long-term sustainability, maintainability, and scalability of software systems. This thesis examines the potential of automated refactoring techniques to enhance code quality and address the challenges posed by evolving software architectures.

The objective of this thesis is to develop and implement an automated refactoring tool within the IntelliJ IDEA integrated development environment. The tool will focus on streamlining codebases, fostering collaboration among development teams, and enhancing the resilience of software systems. The study aims to provide valuable insights for developers and organisations seeking to achieve superior code quality in the context of the rapid evolution of technology.

It is evident that there is a pressing need to comprehend the influence of automated refactoring on code quality, as existing manual methods may not be sufficiently effective in addressing the intricate aspects of modern software architecture.

The methodology entails a comprehensive analysis of existing literature and academic sources with the objective of gaining an understanding of automated refactoring methodologies. An empirical analysis is conducted on existing refactoring plugins, after which a new plugin tailored for the IntelliJ IDEA environment is developed.

The results indicate a notable enhancement in code quality, maintainability, and readability. Empirical analyses have demonstrated the plugin's effectiveness in identifying and correcting code smells and enhancing software structure. The thesis offers practical guidance for developers and organisations on the effective utilisation of automated refactoring tools within their workflows.

Automated refactoring is a highly effective tool for enhancing code quality and addressing the challenges of evolving software systems. This thesis advances the understanding of automated refactoring, contributing to the development of effective tools for maintaining and improving code quality in modern software development practices.

Table of Contents

1 Introduction 1
1.1 Contex........ 1
1.2 Motivation 1
1.3 Objectives........ 1
1.4 Refactoring in Software Development 2
2 Theoretical and Technological Foundations 4
2.1 Overview of Code Quality........ 4
2.1.1 Definition of Code Quality 4
2.1.2 Importance of Code Quality in Software Development 4
2.1.3 Metrics for Assessing Code Quality 5
2.1.4 Contemporary Insights and Advancements 5
2.2 Refactoring Techniques 6
2.2.1 Overview of Common Refactoring Techniques 6
2.2.2 Application and Use Cases in Software Development 8
2.2.3 Empirical Studies on the Effectiveness of Refactoring Techniques 8
2.3 Automated Refactoring Principles........ 9
2.3.1 Definition and Concept of Automated Refactoring 9
2.3.2 Goals and Objectives of Automated Refactoring 9
2.3.3 Benefits and Limitations of Automated Refactoring........ 10
2.3.4 Empirical Studies on Automated Refactoring 11
2.4 Integrated Development Environments (IDEs) 12
2.4.1 Role of IDEs in Software Development........ 12
2.4.2 Overview of IntelliJ IDEA as IDE 13
2.4.3 Integration of Automated Refactoring in IDEs 13
2.5 Analysis of Existing Refactoring Plugins 15
2.6 Personal Empirical Study 16
2.6.1 Surveying Refactoring Practices: Insights from Organizational Engagement........ 16
2.6.2 Impact of Refactoring Initiatives 18
3 Requirements Specification 20
3.1 Requirements Determination........ 20
3.2 Functional Requirements 20
3.2.1 OOP Metrics Calculation 20
3.2.2 Code Smells Detection and Refactoring........ 21
3.2.2.1 Architectural Code Smells 21
3.2.2.2 Implementation Code Smells 22

3.2.2.3 Test Code Smells 23
3.2.3 Customization Options 23
3.3 Nonfunctional requirements 23
3.3.1 Performance Requirements 24
3.3.2 Compatibility Requirement 24
3.3.3 Security Requirements 24
3.3.4 Requirements for Usability 25
3.3.5 Plugin Size Requirements 25
4 Software Design 27
4.1 External components 27
4.1.1 Intellij Code Inspections System 27
4.1.1.1 System Problems Holder 28
4.1.1.2 Program Structure Interface 28
4.1.1.3 Quick Fix 29
4.1.1.4 Element Visitor 30
4.1.2 IntelliJ Threading Layer 30
4.2 Plugin Configuration 31
4.3 Code Inspections Design 33
4.3.1 Detection Mechanism 33
4.3.2 Correction Mechanism 34
4.4 User Interface Design 36
4.5 Customization Menu Design 38
4.6 Algorithmic Design of Components 41
4.6.1 OOP Metrics Calculation 41
4.6.1.1 Lines of Code (LOC) 41
4.6.1.2 Number of Fields (NOF) for classes 41
4.6.1.3 Number of Public Fields (NOPF) for classes 41
4.6.1.4 Number of Methods (NOM) for classes 41
4.6.1.5 Number of Public Methods (NOPM) for classes 41
4.6.1.6 Weighted Methods per Class (WMC) for classes 41
4.6.1.7 Number of Children (NC) for classes 42
4.6.1.8 Depth of Inheritance Tree (DIT) for classes 42
4.6.1.9 Lack of Cohesion in Methods (LCOM) for classes 42
4.6.1.10 Fan-in (FANIN) and Fan-out (FANOUT) for classes 43
4.6.2 Code Smells Detection and Refactoring 43
4.6.2.1 Architectural Code Smells 44
4.6.2.2 Implementation Code Smells 47
4.6.2.3 Test Code Smells 59
5 Project Management 62

5.1 Methodology 62
5.2 Project Planning 63
5.2.1 Scheduling 63
5.2.2 Task Prioritization 64
5.3 Monitoring and Evaluation Plan 68
5.4 Executing the Plan 69
6 Implementation 71
6.1 DevOps Practices 71
6.1.1 Version Control and CI/CD Practices 71
6.1.2 Continuous Integration and Continuous Deployment (CI/CD) ... 71
6.1.3 The Role of Qodana in Ensuring Code Quality 72
6.2 Implemented code 73
7 Testing and Evaluation 76
7.1 Unit Testing 76
7.2 Integration Testing 78
7.3 Manual Testing 80
7.3.1 Functional requirements testing 80
7.3.2 Nonfunctional requirements testing 80
7.4 Beta Testers Review 80
7.4.1 Overview 80
7.4.2 Positive Feedback 81
7.4.3 Places for improvements 82
7.5 Final Evaluation 83
8 User Documentation 85
9 Conclusions and Future Work 87
10 Bibliography 90
11 Annexes 93
11.1 Sorce Code of Implemented Plugin 93
11.2 Tests for Source Code 93
11.3 Test Cases for Manual Testing of Functional Requirements 94
11.4 Test Cases for Manual Testing of Nonfunctional Requirements 104
11.5 Public User Documentation 106
11.6 Plugin Distribution Link 106

1 Introduction

1.1 Contex

In the dynamic and ever-changing landscape of software development, achieving high code satisfaction is critical for ensuring the long-term sustainability, maintainability, and scalability of software program structures. As the era evolves rapidly, there is a growing need for high-quality code. This master's thesis explores automatic refactoring techniques to enhance code satisfaction and address the challenges posed by evolving software structures.

Automated refactoring techniques offer a promising approach to consistently enhance the quality of code while adapting to the changing demands of software development. By automating the method of restructuring code without changing its external behavior, developers can effectively get rid of redundancies, enhance clarity, and simplify complicated code systems. This is not only enables easier maintenance, but, also allows the manner for the seamless integration of recent functions and technology. In this thesis, the ideas and methodologies of automated refactoring are explored, and their impact on code is analyzed within the context of evolving software program systems.

1.2 Motivation

This thesis explores a severe widespread need to understand the highly complicated effects of computerized refactoring on the code quality, bringing about a considerable lack of knowledge in this area. It acknowledges that prevailing manual methods of improving code may not sufficiently address the intricate aspects of modern software architecture. Therefore, research into automated refactoring approaches is in high need to reveal whether they can improve the code quality.

The growing demand for software that not only functions but also maintains, scales, and adapts requires a thorough understanding of how automated refactoring can be effectively used. The goal of this research is to fill in the knowledge gaps by giving valuable inputs for developers, software engineers, and companies to raise their code quality standards.

This thesis aims to present practical implementations and empirical analyses of automated refactoring techniques, moving beyond theoretical issues. The goal is to provide concrete solutions for real-world software development issues. The assumption is that understanding automation refactoring methods can bring a new paradigm of code making. This paradigm aims to ensure that software systems are not only functional but also maintainable, clear, and sustainable in the long term.

In the end this thesis aims at increasing knowledge about automated refactoring and its direct influence on code quality. Using practice, the goal is to bring to the software development community all needed tools and knowledge for dealing with the modern programming landscape. The aim of the thesis is cultivating a culture of continuous refinement of code quality standards in order to promote innovation and superiority in software programming techniques.

1.3 Objectives

The revised objectives of this master's thesis involve the comprehensive understanding, implementation, and evaluation of automated refactoring

principles and methodologies. The central focus is on developing and implementing an automated refactoring tool within the IntelliJ IDEA integrated development environment. The tool aims to streamline codebases, foster collaboration among development teams, and fortify the resilience of software systems. Through empirical analysis and practical application, the thesis seeks to contribute valuable insights that can guide developers and organizations in their quest for superior code quality amid the rapid evolution of technology.

Objectives include:

1. *Comprehensive understanding of automated refactoring principles*:
   a. Undertake in-depth assessment of present literature and academic sources to set up a comprehensive understanding of automated refactoring standards and methodologies.
   b. Analyze numerous automated refactoring strategies employed in software improvement, with a focal point on their application, benefits, and boundaries.
2. *Evaluation of existing refactoring plugins*:
   a. Conduct an intensive analysis of the current panorama of automatic refactoring plugins available across Intellij IDEA IDE.
   b. Evaluate the strengths and weaknesses of popular refactoring plugins, considering factors which include usability and functionality.
3. *Identification of gaps and challenges in existing refactoring tools*:
   a. Identify gaps and challenges in modern-day automatic refactoring through empirical tests and consumer feedback.
   b. Explore limitations in addressing specific issues and their implications for developers and software program development techniques.
4. *Development of a refactoring plugin for IntelliJ IDEA*:
   a. Design and implement a brand-new refactoring plugin specifically tailor-made for the IntelliJ IDEA development environment.
   b. Incorporate capabilities that address recognized gaps in observed refactoring plugins, with a focus on enhancing code quality, maintainability, and readability.
5. *Empirical analysis of the developed plugin*:
   a. Perform comprehensive testing at various levels to evaluate the effectiveness and overall performance of the developed refactoring plugin.
   b. Test drive the plugin with real developers and collect feedback.
6. *Deployment and Documentation for Developers and Organizations*:
   a. Deploy the plugin to the developer community for free use, ensuring wide accessibility and fostering widespread adoption.
   b. Create comprehensive user documentation to assist developers and organizations in effectively integrating automated refactoring practices into their software development workflows.

The purpose of this thesis is to advance the understanding of automated refactoring and to contribute to the improvement of existing tools through the pursuit of the stated objectives.

1.4 Refactoring in Software Development

Automated refactoring techniques are like friendly neighborhood renovators for our code. They are the unsung heroes, quietly operating behind the curtain to ensure our code stays pinnacle-notch. Instead of changing how our software program behaves on the outside, they focus on tidying up the

internal, disposing of useless clutter, making matters simpler to apprehend, and simplifying those pesky, tangled-up code systems.

In today's fast-paced tech world, it is crucial to keep our code in top shape for the long-term success of our software. Automated refactoring provides developers with a useful tool to navigate the ever-changing landscape of software development. This allows them to quickly adapt to new challenges, ensuring that their code remains flexible and robust. This makes renovation easy and allows for future additions of new features and technology.

Automated refactoring is excellent at solving complex issues that manual methods struggle with. Modern software systems can be complex beasts, and conventional strategies simply do not cut it anymore. That is wherein automation steps in, providing an extra state-of-the-art technique to maintain our code in excellent condition. As previously mentioned, there is an increasing demand for software that is not only functional but also easy to maintain, scale, and adapt. And automatic refactoring is key to meeting the ones needs.

To gain a better understanding of automated refactoring, it's essential to examine the tools that enable this process. These plugins are like helpful assistants embedded in our favorite development environments, offering automated solutions to improve code quality.

In the preceding sections, the focus will be on delving into the details of comparing these plugins. Factors such as usability, functionality, and their impact on code quality will be considered. Through this analysis, the aim is to identify strengths, weaknesses, and areas for improvement. Ultimately, the goal is to lay the groundwork for the development of a high-quality refactoring plugin tailored for the IntelliJ IDEA environment.

2 Theoretical and Technological Foundations

2.1 Overview of Code Quality

In this section, an exploration into the crucial aspect of code quality within software development is undertaken. Beginning with a clear definition, the importance of code quality in the creation of robust and efficient software systems is examined. Additionally, various metrics used to evaluate and ensure the excellence of code are explored, along with contemporary insights and advancements in this field.

2.1.1 Definition of Code Quality

A key resource for understanding the subtleties of the idea of code quality is Robert C. Martin's landmark work "Clean Code: Agile Software Craftsmanship" [1]. Martin claims that programming is a way of communication between developers, transcending just the functional correctness of the code. Martin considers clean code as the one with clear syntax, easy to maintain, and following the coding conventions accepted. It is practice-oriented, rather than just principle-guided system development.

This discourse is further augmented by the standpoints given by Bonsignour [2] and Sommerville [3] that complete the picture of code quality. These works argue that quality is the combination of correctness, reliability, efficiency, and maintainability, thus adding a new dimension to the debate on the finer nuances of the code quality concept.

2.1.2 Importance of Code Quality in Software Development

Code quality is an essential component in all stages of software development, starting from the product design to its implementation, distribution, and after-development maintenance. Just as a strong foundation supports a building, a good code constitutes the basis for software so that it can be changed and improved during its lifetime [1]. It is not only about getting the code running, although this is the main goal; it is also about making sure the code is easy to read and collaborate with for all the people taking part in the project because the code is more often being read than it is being written [4].

With time, the importance of clean code for a project becomes even clearer. It serves as the common denominator which all the team members can relate to, thus ensuring smooth collaboration and minimizing confusion. Consider it as telling a very clear story that governs the project advancement, ensuring everybody sticks to the same page.

One of the main benefits of clean code is its positive impact on end-users. Good code allows the software to run very efficiently, giving users a pleasant experience [5]. This means happier users, which in the end translates to the project's huge success.

But clean code represents much more than just practicability; it expresses professionalism and commitment to perfection. It is related to being proud of one's work and constantly working to become a better person.

As such, the role of code quality in software development is vital. It's not all about creating functional code; it is about creating clear code that can be maintained and easy to use for the users. Developers prioritize clean code to contribute to the success of their projects and the satisfaction of their users.

2.1.3 Metrics for Assessing Code Quality

Metrics play an essential function in assessing and enhancing the excellence of software program code. They offer quantitative measurements that enable builders to evaluate diverse components of code quality, which include readability, maintainability, and overall performance. By analyzing these metrics, developers can pick out areas for improvement, prioritize refactoring efforts, and make certain the long-term sustainability of software program systems.

The seminal contribution of Li and Henry (1993) underscores the dynamic nature of code first-rate metrics, transcending their traditional role as static evaluative indices. Their work promotes the use of metrics not only for reviewing past performance but also as proactive tools for continuous improvement in software maintenance management [6]. The iterative nature of software development is emphasised, highlighting the importance of using metrics as guiding principles for continuous improvement and optimisation efforts.

Furthermore, Briand et al. (1996) introduced a unified framework for measuring code quality, contributing to academic discussions by offering a standardized approach to assess various forms of coherence, such as functional and sequential coherence [7]. This framework offers developers a systematic approach to assess organizational and interrelational aspects within a codebase. Functional cohesion measures how closely related the tasks performed by elements within a module are, whereas sequential cohesion evaluates the order and flow of operations within a module. By emphasising cohesion as a crucial factor in code quality, Briand et al. highlight the importance of maintaining logical consistency and readability in software design. Ultimately, this approach fosters the development of robust and maintainable codebases. Additionaly, these metrics help identify potential points of failure or instability in the codebase, allowing teams to mitigate risks early in the development process.

Metrics most commonly used in software development include lines of code (LOC), cyclomatic complexity, code churn, defect density, code coverage, and coupling metrics (such as coupling between objects or services). These metrics offer quantitative measures to evaluate various aspects of code quality, including complexity, maintainability, and reliability [8]. By monitoring and reviewing these metrics throughout the development lifecycle, teams can identify areas for improvement and take proactive steps to enhance code quality and reduce technical debt.

2.1.4 Contemporary Insights and Advancements

As software development methodologies and technologies evolve, the landscape of software engineering continues to be shaped by new insights and advances in analyzing code quality.

One major trend in recent years is the increasing use of automated code review tools and techniques. Machine learning algorithms and advanced static evaluation used to identify issues in code, including bugs, vulnerabilities, and performance bottlenecks, without manual intervention [9]. By integrating automated code evaluation into the development process, teams can identify and address code quality issues early on, reducing the likelihood of defects and improving overall software quality.

Another location of attention is the developing emphasis on code evaluation and collaboration amongst builders. Code evaluation practices, along with pair

programming and code walkthroughs, have been shown to significantly improve code quality by facilitating knowledge sharing, identifying design flaws, and ensuring adherence to coding standards [10]. The rise of distributed version management systems and collaborative development has made code review more accessible and integrated into the development process. This allows teams to maintain high standards of code quality across distributed teams and remote work environments.

Moreover, the emergence of DevOps practices and non-stop integration/non-stop deployment (CI/CD) pipelines has transformed the manner code quality is controlled and monitored during the software program improvement lifecycle. DevOps practices enable teams to identify and address code quality issues early and consistently by automating build, test, and deployment processes. This ensures that only high-quality code is deployed to production environments. Additionally, CI/CD pipelines facilitate the mixing of code high-quality metrics into the improvement workflow, providing real-time feedback to developers and enabling them to make data-driven decisions to improve code quality.

Furthermore, advancements in code quality metrics and measurement techniques continue to refine understanding of code quality and its multidimensional nature. Researchers are exploring new metrics and methodologies to capture components of code best that had been formerly tough to quantify, consisting of code readability, layout complexity, and architectural consistency [11]. By developing extra complete and context-sensitive code great metrics, researchers intend to provide builders with deeper insights into the quality of their codebases and guide them in making greater informed choices to enhance the quality and maintainability of software.

In conclusion, modern insights and advancements in code quality evaluation are pushed by way of a mixture of technological innovation, evolving improvement practices, and ongoing research in software engineering. By leveraging automated analysis equipment, fostering collaboration among developers, embracing DevOps practices, and advancing code quality metrics, corporations can hold to raise their code high-quality standards and supply more reliable and maintainable software solutions.

2.2 Refactoring Techniques

Software engineering involves the practice of refactoring as a fundamental technique. This practice includes various organized techniques aimed at improving the quality, maintainability, and extendibility of existing code while preserving its external behavior. The following section provides a detailed review of the most common refactoring methods, their application, and their importance in software development, as well as an experimental study on their effectiveness.

2.2.1 Overview of Common Refactoring Techniques

Refactoring strategies constitute a powerful toolset for software developers, providing a systematic approach to iteratively upgrade the code design and structure. This chapter covers standard techniques for refactoring, using industry best practices and methods, especially as outlined in “Refactoring.Guru” [12], a leading resource in the software development community. Main refactroring techniques can be splitted on 6 categories:

1. *Composing methods*. This set of techniques focuses on how to organize and structure methods in a codebase. It includes practices such as:

a. Extract method. Involves creating a new method by extracting a section of code from an existing method. The aim is to reduce complexity and improve readability.

b. Inline Method. The opposite of extract method; it involves moving the code inside a method directly into the caller to eliminate unnecessary method calls.

2. Moving features between objects. This category aims at redistributing responsibilities among classes to improve cohesion, reduce coupling, and enhance the overall design of a system. Key methods include:

a. Move method. Transferring a method from its current class to another class where the method is more relevant or where it can better interact with the data it requires.
b. Move field. Similar to moving a method, this involves transferring a field from one class to another to place the field closer to the methods that use it, improving data encapsulation.
c. Extract class. When a class starts doing work that should be done by two or more classes, this technique involves creating a new class and moving the relevant methods and fields from the old class to the new one.
d. Inline class. Involves taking some of the responsibilities of an existing class and moving them to a new class. This is the opposite of Extract Class, where a class does not have enough responsibilities to justify its existence and its features are moved into another class.

3. *Organizing data*. Restructuring the way data is managed and accessed within a software system can improve the clarity, efficiency, and encapsulation of data. Key strategies include:

a. Encapsulate field. By making public fields private and providing access through getter and setter methods, access and modification can be controlled, which helps to encapsulate and increase flexibility.

b. Replace data value with object. Create a new object to provide additional context or functionality if a field or group of fields can be better represented as an object.

4. *Simplifying conditional expressions*. Conditional expressions can quickly become complex and hard to read, making them ripe targets for refactoring. Techniques include:

a. Decompose conditional. Breaking down complex conditional logic into simpler, more manageable methods.

b. Consolidate conditional expression. Combining multiple conditions that lead to the same action into a single conditional statement.

c. Replace nested conditional with guard clauses. Using guard clauses to handle edge cases up front, simplifying the remaining logic.

5. *Simplifying method calls*. Optimizing how methods interact can streamline the flow of information and reduce complexity. Key techniques are:

a. Rename method. Modification of method names to describe their functionality more accurately.

b. Add parameter. The addition of parameters to a method can be an enhancement to its flexibility and utility.

c. Remove parameter. Removes unused parameters to simplify method calls.

6. *Dealing with generalization.* Refactoring towards generalization aims at creating more abstract code structures, allowing for better reuse and flexibility. Techniques involve:

a. Pull up method. Moving similar methods from subclasses to a superclass is a common practice in object-oriented programming. It id sone by identifying common functionality among subclasses, and extracting these shared methods and place them in a superclass.

b. Push down method. In order to avoid unnecessary coupling, unique methods can be moved from a super class to a sub class.

c. Extract interface. The creation of interfaces for classes helps to define clear contracts and support multiple implementations.

2.2.2 Application and Use Cases in Software Development

Refactoring techniques are widely used at various stages of the software development cycle, as explained in "Working Effectively with Legacy Code" by Michael Feathers [13]:

1. *Initial development.* During this phase, developers use methods such as "Extract method" to break down large functions into smaller, more manageable pieces, and "Rename variable/method" to improve code readability. These practices not only make it easier for the development team to understand and debug, but also lay a solid foundation for future enhancements or modifications, while adhering to principles such as SOLID for sustainable software architecture.

2. *Ongoing maintenance.* In the maintenance phase, refactoring is essential to adapt to new business requirements or technology upgrades without breaking existing functionality. Techniques such as "Extract class" help to separate responsibilities for better cohesion, while "Move method/field" can optimise code structure by placing code elements closer to their most relevant context. These practices enable developers to iteratively improve the codebase, ensuring that it remains clean, efficient and scalable.

3. *Legacy codebases.* Dealing with legacy code often involves untangling complex, outdated code structures that impede feature development and bug fixing. Refactoring techniques such as "Inlining" and "Encapsulating fields" are critical to modernising and revitalising these systems. By systematically applying these techniques, developers can significantly reduce technical debt, making code more maintainable and extending its lifespan.

2.2.3 Empirical Studies on the Effectiveness of Refactoring Techniques

Empirical research highlights the effectiveness of refactoring techniques in improving code quality, reducing defects and increasing developer efficiency. These benefits are supported by several studies, each of which provides unique insights into the practice of refactoring within software development:

1. Kim et al. conducted empirical research on a large software program to quantify the effects of refactoring on maintainability and defect rates. The study showed that systematic refactoring practices resulted in a measurable reduction in the number of defects and an improvement in the maintainability score of the software, demonstrating the tangible benefits of refactoring in a real-world, large-scale environment [14].

2. Murphy-Hill et al. explored the perceptions and benefits of refactoring through surveys and analysis of developer activity. The results

showed that developers not only perceived refactoring positively, but also noticed a reduction in the effort required to add new features or fix bugs. This study provided evidence for the role of refactoring in improving developer productivity and software quality from the perspective of those directly involved in software development [15].

3. Ratzinger, Sigmund, and Vorburger's Legacy System Focus examined the impact of refactoring on legacy systems, particularly in terms of reducing technical debt and increasing system adaptability. Their findings highlighted that even small refactoring efforts can lead to significant improvements in the maintainability of the system, providing a pathway for extending the life of legacy systems through targeted refactoring interventions [16].

2.3 Automated Refactoring Principles

Automated refactoring stands at the forefront of modern software engineering practices, offering a systematic approach to improving the structure, maintainability, and quality of software systems. In this section, the foundational principles underlying automated refactoring are explored, including its definition, goals, objectives, as well as associated benefits and limitations.

2.3.1 Definition and Concept of Automated Refactoring

Automated refactoring refers to the process of restructuring existing source code without altering its external behavior [17]. This technique aims to enhance code readability, simplify maintenance, and promote extensibility by systematically applying a series of predefined transformations to the codebase.

Furthermore, automated refactoring facilitates the evolution of software systems over time. As requirements change and new features are added, codebases need to adapt accordingly. Automated refactoring tools streamline this process by enabling developers to make structural changes efficiently and confidently, without the fear of inadvertently breaking existing functionality. This agility is essential in agile software development methodologies, where rapid iteration and continuous improvement are core principles.

At its core, automated refactoring operates based on a set of predefined rules and transformations, often guided by principles from software engineering and design patterns. These transformations range from simple tasks such as renaming variables or extracting methods to more complex operations such as code migration or architectural restructuring. By automating these repetitive and error-prone tasks, developers can focus their efforts on higher-level design and implementation challenges, thus improving overall productivity and software quality.

2.3.2 Goals and Objectives of Automated Refactoring

Besides achieving the primary goals of regular refactoring, automated refactoring tools offer a range of benefits that address specific challenges in the code maintenance lifecycle:

1. *Consistency.* Automated refactoring ensures that changes are applied consistently across the code base, following predefined rules and styles. This consistency helps to maintain a clean, standardised code structure.

2. *Accuracy and security.* Automated tools follow strict algorithms to refactor code, minimising the risk of introducing errors. Many IDEs and refactoring tools also include built-in checks to ensure that refactoring does not change the behaviour of the code.

3. *Cost reduction.* By speeding up the refactoring process and reducing the likelihood of errors, automated refactoring can significantly reduce the costs associated with maintaining and improving code.

4. *Accessibility.* Automated refactoring tools make it easier for developers to apply complex refactoring patterns that they may not be familiar with, thereby improving overall code quality without requiring deep expertise in refactoring techniques.

5. *Documentation and tracking.* Some automated refactoring tools can document changes and provide detailed logs, making it easier to track changes and understand the evolution of the codebase over time.

2.3.3 Benefits and Limitations of Automated Refactoring

Automated refactoring tools bring many benefits to the software development process. However, as with any tool or process, there are limitations and challenges associated with their use. Let's take a closer look at the benefits and limitations of using automated refactoring tools. The benefits include:

1. *Speed and efficiency.* Automated tools can refactor code much faster than manual processes, making it easier to manage large code bases and keep them clean and maintainable.

2. *Detection of hidden problems.* Automated refactoring tools can analyse the entire codebase and detect issues that may not be immediately apparent to developers, such as duplicate code, unused variables, or overly complex methods.

3. *Seamless integration.* Many refactoring tools integrate seamlessly with IDEs and source control systems, providing developers with a smooth workflow and instant access to refactoring operations.

4. *Safe scaling.* Automated tools facilitate the safe scaling of software projects. As codebases grow, so does the complexity and risk of introducing errors during manual refactoring. Automated refactoring tools help manage this complexity and ensure that code remains clean and maintainable as the project scales.

5. *Legacy code integration.* Integrating and updating legacy systems can be challenging. Automated refactoring tools can play a critical role in this process by safely transforming legacy code to meet modern standards and architectures without breaking existing code.

And the limitations of automated refactoring include:

1. *Over-reliance.* There is a risk that developers will become too reliant on automated tools, potentially overlooking the need for a deeper understanding of the codebase, or missing context-specific nuances that require a manual touch.

2. *Initial setup and learning curve.* Implementing automated refactoring tools can require significant setup time and a learning curve for developers unfamiliar with the tool. This process can temporarily slow down development efforts.

3. *Potential for overuse.* Ease of use can lead to over-refactoring, where code is changed more than necessary. This can introduce new bugs or make the code harder to understand and maintain.

4. *Refactoring limitations*. No tool is perfect, automated refactoring tools may not support all programming languages or frameworks equally well, and their ability to detect complex problems or understand contextual nuances may be limited. Detection limitations are mostly related to identifying software design issues.

5. *Cost.* Some of the most sophisticated refactoring tools can be expensive, representing a significant investment for start-ups or smaller teams with limited budgets.

In summary, while automated refactoring offers significant benefits in terms of code quality, maintainability and productivity, its adoption should be considered carefully, considering the specific requirements and constraints of each software project. By understanding the principles, goals and limitations of automated refactoring, development teams can use this powerful technique to drive continuous improvement and innovation in software engineering practices.

2.3.4 Empirical Studies on Automated Refactoring

To broaden the exploration of automated refactoring within software development processes, and to deepen the analysis with more details on research methodologies and additional references, let's delve deeper into the empirical studies and methodologies used to evaluate the impact of automated refactoring tools.:

1. *Thompson and Huang's mixed methods approach* [18]. Thompson and Huang used a mixed methods approach, combining quantitative static code analysis with qualitative interviews, to study the impact of automated refactoring on code quality in open-source projects. They analysed several open-source repositories before and after the application of automated refactoring tools, measuring metrics such as cyclomatic complexity, lines of code (LOC), and code churn. They then conducted interviews with project maintainers to understand the perceived benefits and challenges of incorporating automated refactoring tools into their development workflow. This comprehensive approach allowed them to correlate quantitative improvements in code metrics with qualitative feedback from developers, providing a holistic view of the benefits and limitations of automated refactoring

2. *Longitudinal study by Sanchez and Romero* [19]. In their longitudinal study, Sanchez and Romero tracked the evolution of a large commercial software system over two years, focusing on the adoption and impact of automated refactoring practices. They used a variety of data collection methods, including version control history analysis, issue tracker mining, and semi-structured interviews with the development team. Their findings highlighted not only the immediate benefits of reduced code smells and improved code quality, but also long-term improvements in maintenance effort and developer morale. This study is notable for its length and depth of analysis, providing valuable insights into the lasting benefits of automated refactoring.

3. *Comparative analysis by Nguyen et al* [20]. Nguyen and colleagues conducted a comparative analysis between projects that used automated refactoring tools and those that did not. Using a matched-pair design, they selected projects based on size, domain, and development practices to minimise confounding variables. They then compared several metrics, including bug fix time, feature addition speed, and developer attrition rates, to assess the impact of automated refactoring practices. Their research

provides evidence of the tangible benefits of automated refactoring in improving developer productivity and software quality.

4. *Experimental study by Oliveira and da Silva* [21]. This study involved an experimental setup where development teams were divided into two groups, one using traditional refactoring methods and the other using automated refactoring tools. By monitoring the development process over several iterations, Oliveira and da Silva measured the impact on development speed, bug introduction rates and code complexity. Their results provide empirical evidence of the effectiveness of automated refactoring tools in improving development outcomes, particularly in reducing the time spent on debugging and refactoring tasks.

The empirical studies on automated refactoring provide valuable insights into its benefits and applications in software development. However, they also highlight several gaps and challenges that need to be addressed in future research. These gaps and challenges not only indicate underexplored areas, but also highlight the complexity of effectively integrating automated refactoring tools into software development processes.

1. *Generalizability of result*s. The research observed has focused on specific contexts, such as particular programming languages or open source projects, limiting the applicability of the findings to the diverse landscape of software development. Future studies should aim to cover a wider range of development environments, including different industries, project sizes and languages, to ensure wider applicability.
2. *Lack of long-term effects analysis*. Most studies focus on immediate results, neglecting the long-term effects of automated refactoring, such as maintenance costs and the ongoing management of technical debt. Longitudinal studies that track the longer-term effects of automated refactoring are needed to provide a more complete picture of its benefits and drawbacks.
3. *Lack of cost-benefit analysis*. There's a lack of comprehensive cost-benefit analyses that take into account both the direct and indirect costs associated with adopting automated refactoring tools. Such analyses are essential for organizations to make informed decisions about integrating these technologies into their development processes.

Addressing these areas through targeted research would contribute significantly to the development of more effective automated refactoring practices and tools, facilitating their wider adoption and maximizing their impact on software development.

2.4 Integrated Development Environments (IDEs)

2.4.1 Role of IDEs in Software Development

Integrated Development Environments (IDEs) are essential for software development. They provide a comprehensive set of tools that enhance productivity and optimize the development process. IDEs fulfill several crucial functions:

1. *Advanced code editing.* IDEs provide quite powerful editing tools which allow, for example, syntax highlighting and code completion, thus coding process gains in efficiency.

2. *Efficient debugging*. Debuggers embedded in IDEs help developers easily locate and fix bugs, improving code quality and stability.

3. *Seamless version control integration*. IDEs smoothly work with version control systems making synchronous collaboration easy and managing code repositories across teams straightforward.

4. *Automated build processes.* IDEs automate the build process therefore they simplify the associated compilation and execution tasks thus freeing developers time and effort.

5. *Automated code refactoring.* IDEs facilitate automated code refactoring; hence developers can easily adjust the structure, readability, and maintainability of code.

6. *Testing support.* IDEs help write and run tests which in turn allow for early detection and resolution of defects ensuring smooth software that can endure.

7. *Streamlined deployment*. IDEs simplify software deployment by automating the packing and deployment of software artifacts to various environments with minimal human intervention.

In summary, IDEs provide software developers with a comprehensive set of tools and features that simplify the complexities of programming. From code suggestions and debugging capabilities to code navigation and version control integration, IDEs are invaluable assets that significantly enhance the development process.

2.4.2 Overview of IntelliJ IDEA as IDE

IntelliJ IDEA is an integrated software development environment for many programming languages, in particular Java, JavaScript, Python, developed by JetBrains [22].

The design of the environment is focused on programmers' productivity, allowing them to concentrate on functional tasks, whereas IntelliJ IDEA takes care of routine operations. Starting with version 9.0, the environment is available in two editions: Community Edition and Ultimate Edition. Community Edition is a completely free version available under the Apache 2.0 license, with full support for Java SE, Groovy, Scala, and integration with the most popular version control systems. The Ultimate Edition, available under the commercial license, supports Java EE, UML-diagrams, as well as support for other version control systems, languages, and frameworks.

Supported languages for last version (for 4th of June 2024) of IntelliJ IDEA 2024.2 includes : Java, JavaScript, CoffeeScript, HTML/XHTML/HAML, CSS/SASS/LESS, Python, Ruby, Haxe, Groovy, Scala, SQL, PHP, Kotlin, Clojure, C, C++, Go and others.

2.4.3 Integration of Automated Refactoring in IDEs

IDEs incorporate refactoring features into the code editor through automation facilitating developers to perform elaborate code restructuring easily. These features typically include:

1. *Refactoring menu.* IDEs usually come with a dedicated menu or toolbar for accessing different refactoring options. This menu usually provides a comprehensive list of all available refactorings, for example renaming variables, extracting methods, and restructuring code.

2. *Contextual refactoring suggestions.* IDEs, by analyzing the context of the code, offer intelligent hints for possible refactorings. These

suggestions can appear as inline tips or appear as command prompts when code patterns were detected, promoting proactive handling of poor code quality.

3. *Refactoring wizards.* For more complicated refactorings the IDEs may provide interactive wizards or dialogs to walk developers through the process in a step-by-step manner. The wizards give options to customize the refactoring behavior as per the developers' choices.

4. *Real-Time feedback*. IDEs give feedback in real time during refactoring operations to show you the possible problems or conflicts which can occur. Through this kind of feedback, the developers will be able to make informed decisions and the codebase will be kept intact all throughout the course of refactoring.

IDEs typically provide a rich ecosystem of plugins and extensions that enable developers to extend their refactoring abilities using third-party tools and integrations. These plugins can give specific refactorings, integrate with external code analysis tools, or more automation features. Some common ways in which plugins enhance refactoring in IDEs include:

1. *Extended refactoring catalog*. Plugins can enrich the selection of refactorings available beyond the native functionality of the IDE. Developers can install plugins to access language- and framework-specific refactorings.

2. *Integration with external tools*. The modules provide smooth integration with external code analysis and transformation utilities. This integration allows developers to use the specialized tools' strength within the comfortable milieu of their IDEs, which boosts the efficiency and quality of the code.

3. *Custom refactoring workflow.* Plugins empower developers to create custom refactoring workflows tailored to their specific development practices. Developers can combine multiple refactorings into automated sequences or define custom rules for code restructuring, optimizing their development process.

IDEs integrate with other development tools and platforms to automate the refactoring process and make the code follow the same guidelines throughout. Key aspects of this collaboration include:

1. *Version control integration.* IDEs are seamlessly integrated with version control systems, thereby enabling developers to perform refactorings within the scope of a collaborative code repository. Features of version control like branching, merging, and code reviews are closely linked with refactoring workflows, making sure that code changes are handled properly.

2. *Build automation.* IDEs work with build automation tools to guarantee that the refactoring changes are easily incorporated into the build and deployment pipeline. Automated build processes monitor and propagate refactoring changes, limiting integration errors, and upholding the continuity of the development cycle.

3. *Continuous integration/Continuous deployment (CI/CD).* IDEs integrate with CI/CD platforms to automate the validation and deployment of refactored code. Continuous integration pipelines are configured to trigger automated tests and deployments in response to refactoring events, enabling rapid feedback and continuous delivery of high-quality software.

In summary, through the integration of refactoring features, utilization of plugins, and teamwork with other developers' tools, IDEs create a complete platform for optimization of the refactoring process and enrichment of the software development ecosystem.

2.5 Analysis of Existing Refactoring Plugins

The analysis of analogs preceding the implementation is necessary since it provides awareness of the full picture of the existing implementations, enabling to discovery of the weak and strong aspects. Through studying analogous projects or products, previous experiences can be leveraged to inform decisions, avoid common mistakes, and utilize effective strategies. This approach improves performance and increases the likelihood of achieving expected results. Analog analysis serves as a valuable learning tool that stimulates innovation and appropriate decision-making during project implementation.

In the realm of refactoring tools, three standout plugins are Refactor-J, CodeGlance, and SonarLint. These tools are renowned for their ability to streamline code optimization and improve code quality.

1. Refactor-J:

   Positive Aspects:

   a. Rich refactoring options. Refactor-J offers a diverse array of refactoring options, from simple variable extractions to complex method extractions, making it a comprehensive tool for various scenarios.
   b. Strong community support. With an active community, Refactor-J benefits from continuous feedback and improvement suggestions, ensuring a responsive development cycle.
   c. Code analysis features. It provides intelligent code analysis features, aiding developers in identifying potential refactoring opportunities, thereby enhancing code quality.

   Negative Aspects:

   a. Learning curve. Due to its extensive feature set, Refactor-J has a steeper learning curve, potentially challenging for novice developers.
   b. Limited customization. While robust in built-in features, the plugin lacks extensive customization options, limiting adaptability to specific project needs.

2. CodeGlance:

   Positive Aspects:

   a. Code navigation aid. CodeGlance offers a code minimap, providing developers with a visual aid for efficient code navigation and orientation within large files, enhancing productivity.
   b. Sleek interface. Its minimalistic design integrates seamlessly into IntelliJ IDEA's interface, ensuring an unobtrusive user experience.
   c. Lightweight. CodeGlance operates efficiently, consuming minimal system resources, ensuring smooth performance even on less powerful development machines.

Negative Aspects:

a. Limited refactoring capabilities. CodeGlance primarily focuses on code visualization and navigation, lacking robust refactoring features, making it unsuitable for in-depth code restructuring needs.

b. Dependency on IDE updates. It heavily relies on IntelliJ IDEA updates; if not promptly updated, compatibility issues may arise, impacting its functionality.

3. SonarLint:

Positive Aspects:

a. Good code quality analysis. SonarLint excels in code quality analysis, providing real-time feedback on code issues, security vulnerabilities, and coding standards violations, aiding developers in writing cleaner code.

b. Integration with SonarQube. Seamless integration with SonarQube, a popular code quality management platform, allows for centralized code analysis, fostering collaborative code quality improvement efforts.

c. Support for multiple languages. The tool supports multiple programming languages, making it versatile for projects involving diverse technologies.

Negative Aspects:

a. Limited refactoring features. SonarLint primarily focuses on static code analysis and lacks advanced refactoring capabilities, limiting its utility in actual code restructuring tasks.

b. Dependence on third-Party services. Integration with SonarQube may establish dependencies on third-party services, possibly affecting performance and security in particular development settings.

2.6 Personal Empirical Study

2.6.1 Surveying Refactoring Practices: Insights from Organizational Engagement

In order to gauge the practical implementation and understanding of refactoring techniques in an organizational context, a survey was conducted among 16 individuals, ranging from junior developers to architects, within the company where I am currently working. The aim was to determine the prevailing attitudes towards and frequency of refactoring practices among software development professionals.

The survey commenced by inquiring about the frequency of manual refactoring practices during the participants work processes. Encouragingly, 100% of the respondents affirmed their engagement in refactoring endeavors. A breakdown of the responses revealed that the majority, constituting 68.8% of participants, regularly integrate refactoring into their work routines. Meanwhile, 31.3% perform refactoring either occasionally or as per specific needs or requests. Results presented on Figure 2.1.

Do you practice refactoring during your work process?

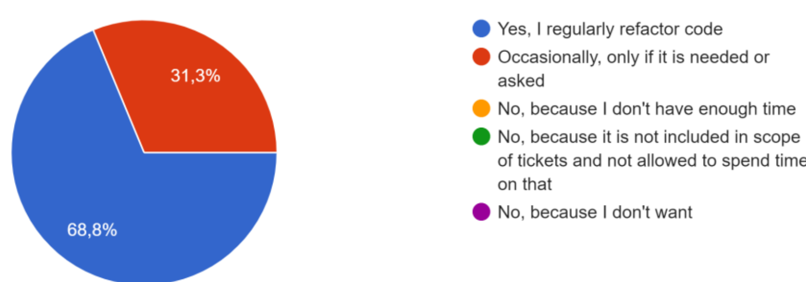


Figure 2.1 - Frequency of refactoring practices

Also, participants were then asked to describe how much time they spend refactoring both legacy and new code each month. Notably, the responses highlighted varying degrees of commitment to refactoring efforts:

1. *Refactoring legacy code.* When it comes to refactoring existing codebases, a diverse distribution of time allocation emerged. A significant proportion, 37.5% of respondents, reported spending between 4 and 8 hours per month on this endeavor. Meanwhile, 25% invest between 2 and 4 hours and a further 25% allocate 8 hours or more. A modest 12.5% spend less than 1 hour on this activity. Results presented on Figure 2.2.

How many hours per month you spend on refactoring old code?

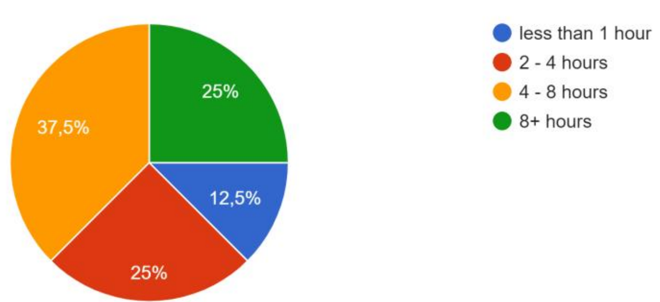


Figure 2.2 - Time allocation for old code refactoring

2. *Refactoring new code.* Responses were similarly mixed when it came to optimizing recently written code. A substantial 37.5% reported spending between 2 and 4 hours per month on this task. Similarly, 37.5% spend less than an hour on this activity. However, there is a notable difference when compared to legacy code, with only 12.5% spending between 4 and 8 hours and a further 12.5% allocating 8 or more hours per month. Results presented on Figure 2.3.

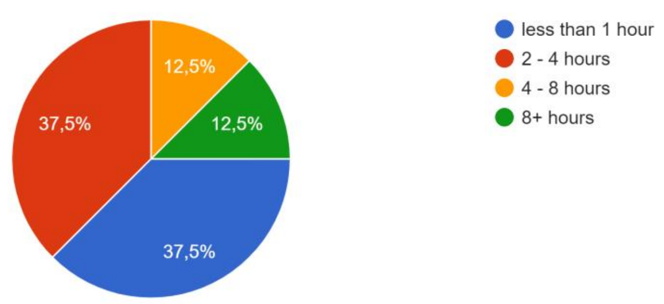


Figure 2.3 - Time allocation for new code refactoring

The results of the survey underline the importance of optimizing the use of time in software development. The significant amount of time spent on manual refactoring activities highlights the potential for significant efficiency gains through the adoption of automated refactoring tools. The adoption of automation will improve development processes and foster a culture of innovation, productivity, and continuous improvement within the organization.

2.6.2 Impact of Refactoring Initiatives

This section explores the tangible results of a significant manual refactoring my team performed on the current project at work in May 2023. The refactoring was carried out by 2 people in a two-week period (one sprint). The subsequent improvements in the development speed and code quality metrics serve as compelling evidence of the effectiveness of our refactoring efforts.

In order to improve the maintainability, scalability and overall quality of our code base, my team embarked on a major refactoring initiative in May 2023. This effort involved meticulously restructuring and optimising the existing code. The overall goal was to provide a more robust foundation for ongoing development activities. The refactoring included design fixes as well as finding and getting rid of any code smells that might be causing potential problems.

Following the completion of the refactoring initiative, a significant improvement in development velocity was observed. Before the initiative, our team typically achieved around 35 story points per sprint. However, after the refactoring initiative, this figure jumped to around 53 story points per sprint. This significant increase in productivity (on 50%) underlines the positive impact of refactoring on our ability to deliver value more efficiently and effectively. This means that code refactoring improves maintainability and makes it easier to add new functionality.

In addition, the refactoring initiative resulted in significant improvements in various code quality metrics, as evidenced by the analysis of the “SonarQube” report. A comparative assessment of the pre and post refactoring metrics shows a marked improvement in several key areas, the results of which are shown in Figure 2.4.

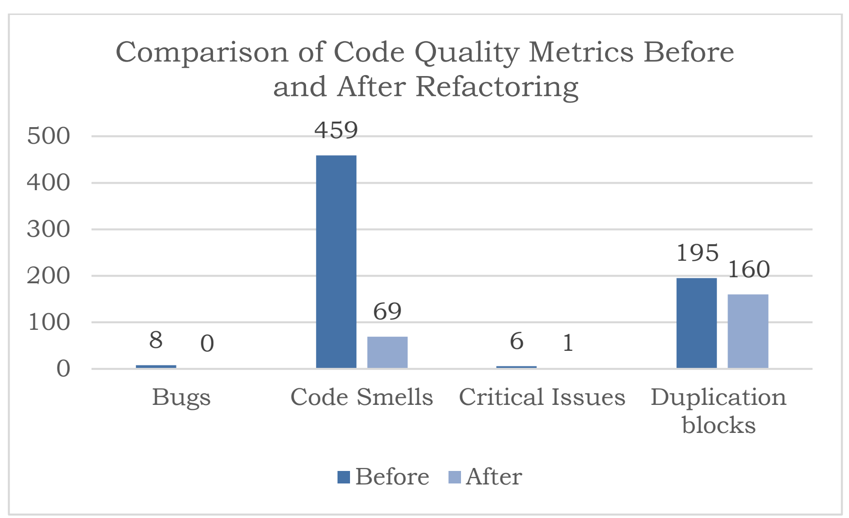


Figure 2.4 – Diagram “Comparison of Code Quality Metrics Before and After Refactoring”

The drastic reduction in the number of bugs, code smells, critical issues and duplicate blocks underlines the success of our refactoring efforts in improving the overall quality and robustness of our codebase. These improvements not only contribute to a more stable and reliable software product, but also facilitate smoother development processes and easier maintenance in the long term.

The results of our refactoring initiative illustrate the significant benefits that can be achieved through proactive investment in code optimisation and maintenance. The observed acceleration in development velocity and notable improvements in code quality metrics serve to validate the effectiveness of refactoring as a strategic software engineering practice. Going forward, these findings will inform our approach to codebase management, emphasising the importance of periodic refactoring activities in maintaining and improving the quality and efficiency of our software development efforts.

3 Requirements Specification

3.1 Requirements Determination

The development of any software project requires a thorough understanding of the requirements that drive its design and implementation.

The requirements gathering process began with a thorough analysis of the refactoring capabilities built into IntelliJ IDEA. This initial stage was important to ensure that efforts were not redundant and implemented functions not duplicating existing functionality.

Next, I delved into a detailed evaluation of existing refactoring plugins. The analysis carried out in chapter 2.5 made it possible to identify the strengths and weaknesses inherent in the various solutions available on the market. Based on the insights gained from these evaluations, I endeavored to incorporate the best practices and innovative approaches observed in these plugins into my own solution.

In parallel with the technical exploration, it is very important to work with domain experts. A key milestone in this journey was collaboration with the solution architect at “Hapag-Lloyd”, a leading global shipping company [23]. Through consultations and knowledge-sharing session, I gained invaluable insights into the specific requirements and challenges associated with code refactoring in their projects. These interactions not only enriched understanding of the domain, but also guided the refinement of the plugin to meet the unique needs of enterprise-level software development.

Furthermore, the requirements creation process was enriched by leveraging first-hand experience from code review activities. As an active participant in the code review process, I carefully documented the recurring corrections and improvements suggested to the developers. These insights provided a wealth of empirical data on common pitfalls and areas for improvement in the codebases. Using these corrective actions, I was able to distil the essential requirements for the refactoring tool to ensure that it effectively addressed common issues encountered during code reviews.

In summary, the journey towards defining the requirements for the refactoring tool plugin was characterized by a multifaceted approach that included technical analysis, comparative evaluation, expert consultation, and empirical insights. By synthesizing these diverse perspectives, I have tried to develop a set of requirements that meet the needs of developers and address the intricacies of modern software development practices.

3.2 Functional Requirements

3.2.1 OOP Metrics Calculation

1. *Calculate Lines of Code (LOC).* The plugin should calculate the number of lines of code within classes to provide insights into code complexity and size.
2. *Determine Number of Fields (NOF) for classes.* The plugin should count the number of fields within each class, providing insights into class complexity and potential cohesion issues.
3. *Compute Number of Public Fields (NOPF) for classes.* The plugin should count the number of public fields within each class, helping developers assess encapsulation and information hiding principles.

4. *Calculate Number of Methods (NOM) for classes.* The plugin should count the number of methods within each class to provide insights into class complexity and cohesion.

5. *Determine Number of Public Methods (NOPM) for classes.* The plugin should count the number of public methods within each class, allowing developers to assess class interface complexity and encapsulation.

6. *Calculate Weighted Methods per Class (WMC) for classes.* The plugin should calculate the weighted sum of methods within each class, considering factors such as method complexity or importance, to assess class complexity.

7. *Compute Number of Children (NC) for classes.* The plugin should calculate the number of immediate subclasses for each class in the inheritance hierarchy, providing insights into class hierarchy and potential design issues.

8. *Determine Depth of Inheritance Tree (DIT) for classes.* The plugin should calculate the maximum length of the inheritance path from each class to the root of the inheritance hierarchy, helping developers assess class complexity and inheritance depth.

9. *Calculate Lack of Cohesion in Methods (LCOM) for classes.* The plugin should calculate the lack of cohesion in methods for each class, providing insights into class design quality and potential refactoring opportunities.

10. *Compute Fan-in (FANIN) for classes.* The plugin should calculate the number of classes that depend on (use) a particular class, helping developers identify classes with high coupling and potential design issues.

11. *Compute Fan-out (FANOUT) for classes.* The plugin should calculate the number of classes that a particular class depends on (uses), providing insights into class dependencies and potential design issues.

3.2.2 Code Smells Detection and Refactoring

Effective software development necessitates identifying and addressing code smells — indicators of potential design or implementation flaws that can impede code readability, maintainability, and extensibility. The developed plugin will provide comprehensive detection capabilities across architectural, implementation, and test-related code smells, facilitating targeted refactoring efforts for improved code quality.

3.2.2.1 Architectural Code Smells

1. *Detect cyclic dependency between modules or components.* The plugin will analyze import and dependency relationships between modules or components, identifying instances where circular dependencies hinder code maintainability and scalability. It will provide visual cues or reports highlighting these dependencies for developers to prioritize refactoring efforts effectively.

2. *Identify God components.* By analyzing code complexity metrics and coupling between classes or modules, the plugin will flag components with disproportionately high complexity or functionality as potential "God Components". Developers will receive actionable insights to refactor these components into smaller, more cohesive units, thus improving code comprehensibility and maintainability.

3. *Identify scattered functionality.* Through code analysis, the plugin will identify instances where related functionalities are dispersed across disparate modules or classes. It will pinpoint these scattered functionalities, enabling developers to consolidate them logically, enhancing code organization and reducing redundancy.

3.2.2.2 Implementation Code Smells

1. *Detect reference object comparison instead of content comparison.* The plugin will inspect object comparison operations occurrences where reference comparison is used instead of `Object.equals` It will suggest replacing such instances with proper `equals` comparisons, ensuring consistent and reliable object equality checks.

2. *Detect content enums comparison instead of reference comparison.* By analyzing enum comparisons, the plugin will identify cases where `equals()` or `Objects.equals` is utilized instead of ==. It will provide refactor suggestions to ensure correct comparison of enum values by reference, promoting code correctness and avoiding potential bugs.

3. *Detect and refactor complex conditional statements.* Utilizing static code analysis, the plugin will identify convoluted conditional statements (more than 4 conditions) that hinder code readability and maintainability. It will offer refactoring recommendations to simplify these conditions, making the code more comprehensible and easier to maintain.

4. *Boolean Expression Optimization.* The plugin should provide optimization capabilities for boolean expressions within the codebase. This optimization should involve simplifying complex boolean expressions to improve readability and efficiency.

5. *Object method parameters check.* The plugin should examine method calls involving objects and confirm whether only specific properties are used within the method. If the method accesses only one property from an object, it should be revised to accept that property alone as a parameter, rather than the entire object.

6. *Identify and refactor long methods.* Through code metrics analysis, the plugin will identify lengthy methods exceeding predefined thresholds. It will suggest breaking down these methods into smaller, more focused ones to enhance code modularity and understandability while adhering to the Single Responsibility Principle (SRP).

7. *Method naming conventions.* The plugin should encourage proper naming conventions for methods based on the action they perform. So, tool will provide intelligent suggestions for method names based on DDD purposes:

   a. *Validating.* Method should have “validate” prefix: `validateOrder(), validateUserCredentials();`

   b. *Updating.* Method should have “update” or “set” prefix: `updateInventory(), updateCustomerDetails();`

   c. *Creating.* Method should have “craete” or “generate” prefix: `createProduct(), createUser();`

   d. *Deleting.* Method should have “delete” prefix: `deleteRecord(), deleteCartItem();`

   e. *Retrieving.* Method should have “get”, “find”, “select”, “list” or “retrieve” prefix: `getCustomerById(), retrieveOrders();`

   f. *Processing.* Method should have “process” prefix: `processPayment(), processOrder();`

   g. *Handling.* Method should have “handle” prefix: `handleException(), handleEvent();`

8. *Detect and extract to constants magic numbers and strings.* By examining both numeric literals and string literals within the codebase, the plugin will identify magic numbers and magic strings lacking explanatory context. It will propose replacing these magic numbers and strings with named constants,

improving code clarity, enhancing maintainability, and reducing the likelihood of errors during maintenance.

9. *Extract constants from code.* Through pattern recognition, the plugin will identify recurring objects within the code and recommend extracting them as constants. This practice enhances code maintainability, reduces redundancy, and facilitates future updates or modifications.

10. *Calculate and refactor cyclomatic complexity.* The plugin will calculate the cyclomatic complexity of methods, offering insights into their structural complexity. It will generate reports or visualizations highlighting methods with high complexity (level upper than 6 by default), aiding developers in identifying areas for refactoring to improve code maintainability and testability.

3.2.2.3 Test Code Smells

1. *Enforce test naming convention.* The plugin will enforce a naming convention for test methods as specified by the user in the settings, ensuring clarity and coherence in test case descriptions. This customization will facilitate a better understanding of the test cases and their associated scenarios.

2. *Identify and handle empty and ignored tests*. Through test suite analysis, the plugin will detect empty test methods that do not contain any assertions or test logic. It will provide notifications or recommendations to either populate these tests or remove them to maintain an effective test suite.

3. *Ensure parameters for tested methods have prefixes "given" or "actual".* The plugin will enforce naming conventions for test parameters, ensuring consistency and clarity in test method signatures. This practice enhances the readability and maintainability of test code, aiding developers in understanding test scenarios.

4. *Ensure asserted result variables have prefix "expected".* By analyzing test assertions, the plugin will ensure that asserted result variables are appropriately labeled with the expected prefix. This standardization improves the readability and comprehensibility of test assertions, facilitating easier debugging and maintenance.

3.2.3 Customization Options

1. Ability to configure thresholds for code smell detection. The plugin should allow users to customize thresholds for detecting code smells, enabling flexibility based on project-specific requirements and coding standards.

2. Option to enable/disable specific refactoring rules in a separate window. The plugin should provide users with the option to enable or disable specific refactoring rules based on their preferences and project needs, allowing for a tailored refactoring.

3.3 Nonfunctional requirements

This section defines the non-functional requirements for the proposed IntelliJ IDEA plugin, including performance, compatibility, security, usability, customization constraints, and plugin size. These criteria serve as key standards for the plugin's functionality, efficiency, and usability. Before the plugin is made available, these prerequisites will be carefully examined and verified to make sure it satisfies all requirements.

3.3.1 Performance Requirements

Performance is an important aspect of any software tool-especially for automated refactoring tools like IntelliJ IDEA. Users expect plugins to improve their productivity without introducing significant overheads or slowdowns. Establishing clear performance requirements is therefore essential to ensure a smooth and efficient user experience.

1. *Responsiveness*. The plugin must respond to user interactions within 5 seconds for codebases of up to 10,000 lines of code. This constraint is critical as delays beyond this threshold can have a noticeable impact on developer productivity and engagement. Developers often expect immediate feedback when performing refactorings on smaller codebases in order to maintain workflow momentum.
2. *Memory usage.* The plugin must operate within a memory limit of 1GB of RAM. This constraint is particularly important for developers working in resource-constrained environments, such as low-end development machines or shared development servers. Excessive RAM usage can lead to performance degradation, system instability or even crashes, negatively impacting the overall user experience.
3. *Plugin Interactions.* The plugin should interact seamlessly with other IntelliJ IDEA plugins and extensions, ensuring that combined functionality does not introduce conflicts or performance issues. Compatibility testing should include scenarios where multiple plugins are active simultaneously to identify and resolve any compatibility or performance-related issues proactively.

3.3.2 Compatibility Requirement

Compatibility guarantees that the plugin works seamlessly with existing development environments and technologies, increasing its usability and acceptance. This section goes over compatibility factors, such as platform support and version compatibility.

1. *IDE Compatibility.* The plugin should be compatible with IntelliJ IDEA 2021.1 and later (current latest version 2024.2) Compatibility with the most recent IDE releases means that users may take use of the plugin's capabilities and additions without facing compatibility difficulties or limits inherent in previous software versions.
2. *Language support.* The plugin shall support Java 8 and above, accommodating developers using modern Java language features. Developers can take advantage of features such as lambda expressions, streams, and default methods provided by Java 8 and above. This ensures that the plugin is compatible with the latest advancements in Java programming.

3.3.3 Security Requirements

The security of user data is paramount in the development and implementation of plugin. Commitment extends to ensuring that no information is gathered or communicated outside of the local development environment. Below are the measures undertaken to protect sensitive user data and code:

1. *Data security.* Plugin only collects essential information required for its functionalities. Gathering any extraneous data should be avoided to minimize the risk of exposure.

2. *Local environment restriction.* All data processing and storage occur within the local development environment. This means that user credentials, code samples, project metadata, and any other sensitive information are kept strictly within the user's system. No usage of external systems is allowed.

3.3.4 Requirements for Usability

Improving user satisfaction and streamlining development processes are key goals of the usability enhancements for the plugin. This section specifies requirements related to user interface design, documentation accessibility, and providing clear and actionable refactoring suggestions. These requirements aim to ensure that users can easily understand and use the plugin's features, make informed decisions during refactoring tasks, and access relevant documentation for guidance and troubleshooting.

1. *Documentation.* Easy-to-read documentation should be created to help users use the features and functions of the plugin. To do this, the plugin will offer readily available documentation via a variety of platforms, including as web documentation sites and the integrated help system in IntelliJ IDEA. A wide range of subjects will be covered in the documentation, such as best practices, installation instructions, usage recommendations, and troubleshooting advice.
2. *User-Friendly Refactoring Recommendations.* The plugin should provide clear and actionable suggestions for refactoring, highlighting potential issues within the codebase. When hovering over identified issues, a dialog should display concise explanations and offer correction options, guiding users through the refactoring process in a straightforward manner.
3. *Project Analysis and Code Smells Display.* The plugin should provide full project analysis capabilities, including object-oriented programming (OOP) metrics, to give users insight into the structure and complexity of their codebase. This analysis should identify and display all code smells and potential improvement areas related to implementation, architecture and test code. Code smells should be presented in a separate window within IntelliJ IDEA, allowing users to systematically review and address them. The presentation of code smells should include visual information such as diagrams and pie charts that categorise code smells by type (implementation, architecture, test) to provide users with a clear overview of areas requiring attention and potential refactoring efforts. This visual representation helps to prioritise refactoring tasks based on the severity and impact of code smells on code quality and maintainability.

3.3.5 Plugin Size Requirements

The installation package for the plugin must be streamlined to reduce size without sacrificing important features and functionality. Therefore, to minimize the total package size, the plugin will use resource bundling, code minification, and compression techniques. In addition, unnecessary or obsolete components must be removed or deprecated to decrease the footprint. The plugin installation package's target size cannot be more than 10MB, what ensures ensures efficient download and installation processes, especially in environments with limited bandwidth or resources. This size restriction encourages the use of optimisation techniques such as resource bundling and code minification while maintaining essential features and functionality. It also helps to speed up updates and

ensure compatibility with different system configurations without overwhelming the user or the system's memory capacity.

4 Software Design

4.1 External components

4.1.1 Intellij Code Inspections System

The IntelliJ Code Inspections System [24], an integral part of IntelliJ IDEA, serves as a comprehensive code analysis tool that identifies errors, inefficiencies, and potential improvements for source code. This system works dynamically, continuously inspecting code as it is written and making suggestions for improvements and fixes (Figure 4.1).

```
1  public class DeepDataFlow7 {
2      void process() {
3
4          Object first = "hello";
5          Object second = 123;
6
7          if (first.equals(second)) {
8              System.ou   'equals' between objects of inconvertible types 'String' and 'Integer'
9          }               java.lang.Object
10                         @Contract(value = "null -> false", pure = true)
11                         public boolean equals(
12     }                       Object obj
13 }                       )
```

Figure 4.1 - Example of IntelliJ Code Inspections System usage

The IntelliJ code inspections system plays a key role in the development of refactoring plugins. And while IntelliJ IDEA provides built-in refactoring capabilities, custom refactoring plug-ins are often developed to extend these capabilities and address specific project requirements.

When developing refactoring plugins, the IntelliJ code inspections system is used to ensure the reliability and correctness of code transformations. Prior to performing a refactoring operation, the plugin uses the Inspections system to perform a comprehensive analysis of the code base. This analysis includes identifying potential conflicts, dependencies and side-effects that may result from the proposed refactorings.

Developers can create their own inspections in plugins by inheriting from abstract inspection classes provided by IntelliJ IDEA's API. By extending these classes, developers gain access to a wide range of functionality for defining custom inspections tailored to specific code patterns or project requirements. This approach allows developers to implement sophisticated code analysis algorithms and seamlessly integrate them into their refactoring plugins. The interaction with the IntelliJ Inspections System has various benefits:

1. *Asynchronous execution.* Code inspections take place in the background, paralleling user activities with the IDE. This functionality separates the difficulties of maintaining worker and UI threads, allowing for more concentrated attention to code smell detection.
2. *Automated triggering.* Inspections are triggered automatically if code is modified. Several improvements, such as the Local Inspection Tool, improve efficiency by restricting inspections to the currently open file and rerunning them only on updated code blocks. This simplified technique shortens the feedback loop and provides developers with timely information of inspection findings.
3. *Visual feedback.* The plugin can use the code highlighting capability to visually highlight code blocks that contain recognized code smells, while also showing normal warning highlights to notify users.

Furthermore, by extending the Inspections System, developers may handle code smell inspections alongside other built-in inspections, including the ability to suppress warnings, customize inspection severity, and deactivate particular inspections.

But also has certain disadvantages as well:

1. *Platform Dependency.* The plugin's architecture will be strongly dependent on IntelliJ IDEA, making it intrinsically incompatible with other IDEs such as Eclipse or Visual Studio owing to basic architectural incompatibilities.
2. *Limited Flexibility.* While the Inspections System improves code checks, it limits the ability to adopt custom optimization algorithms. For example, eliminating needless repeats of inspections against irrelevant code modifications is difficult owing to the system's intrinsic constraints.

4.1.1.1 System Problems Holder

The IntelliJ IDEA System Problems Holder [25] is an essential part of managing detected code issues within the IDE. It acts as a centralised repository for storing information about various problems, inconsistencies and violations identified by the code inspection system. These problems can range from simple syntax errors to complex code quality issues and are categorised based on severity levels such as errors, warnings and suggestions. The System Problems Holder organises these problems according to their location within the source code, providing developers with a structured and accessible interface for reviewing and resolving detected problems.

One of the key benefits of using System Problems Holder is its seamless integration with the IntelliJ IDEA user interface. Developers can interact with detected code problems directly from within the IDE, accessing context menus, tooltips, and navigation shortcuts to explore problem details and apply corrective actions. This tight integration allows users to streamline the refactoring process by easily navigating to problematic code locations, reviewing suggested fixes, and applying refactorings with minimal effort.

To register issues in the System Problems Holder while developing a refactoring plugin for IntelliJ IDEA, developers create custom code inspections to analyse the codebase for specific issues. They implement inspection logic to detect these issues and report them using `ProblemDescriptor` objects. These descriptors are then registered with the System Problems Holder, making them visible to users within the IDE. Optionally, developers can provide quick fixes to address the detected issues efficiently.

4.1.1.2 Program Structure Interface

The Program Structure Interface (PSI) [27] in IntelliJ IDEA presents a unified model for accessing and manipulating various programming language constructs. It covers a wide range of elements, including classes, methods, variables, and statements. The PSI Tree is a hierarchical representation of code elements within the PSI, organized in a tree-like structure that reflects the relationships between different elements.

The Program Structure Interface (PSI) in IntelliJ IDEA presents the abstract syntax tree (AST) of code within the IDE. The PSI acts as an interface between the IDE's underlying code model and the user interface, enabling

powerful code analysis, navigation, and refactoring capabilities. The PSI Tree represents the PSI and emphasizes the hierarchical structure of code elements within the AST. Each node in the PSI Tree corresponds to a specific code element, reflecting the syntactic structure and dependencies of the code. The PSI Tree offers a clear and concise view of the hierarchical organization of code elements, making it ideal for operations such as code navigation and analysis, example shown at Figure 4.2.

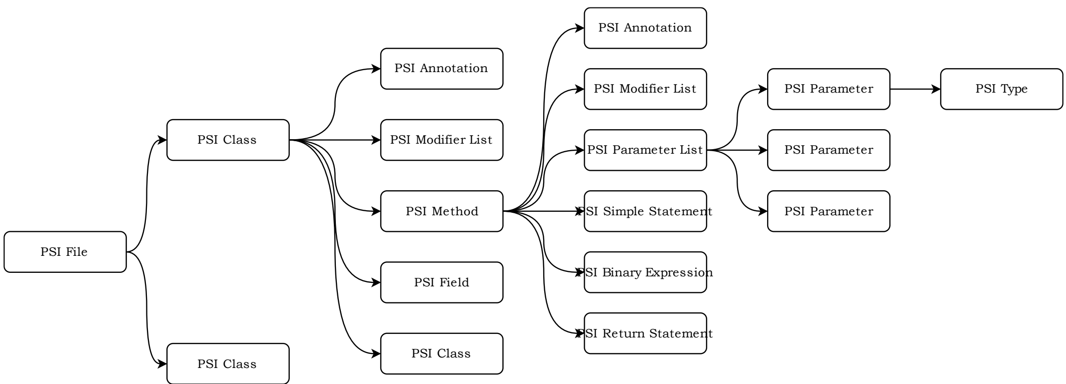


Figure 4.2 - Example of PSI Tree

The PSI empowers developers to write robust tools and plugins for code editing and analysis by abstracting the syntactic structure of code into a unified model, enabling powerful code manipulation and introspection capabilities.

The PSI Tree encompasses a wide range of language-specific elements, with each element represented as a node. For Java, these elements include:

1. *Classes and Interfaces.* Representations of class and interface declarations, including their modifiers, methods, and fields.
2. *Methods and Functions.* Definitions of methods, functions, and other executable code blocks, along with their parameters and return types.
3. *Variables and Fields.* Declarations of variables, fields, and constants, including their types, names, and initialization expressions.
4. *Statements and Expressions.* Instances of language-specific statements and expressions, such as if statements, loops, assignments, and method invocations.
5. *Annotations and Modifiers.* Annotations and modifiers applied to code elements, influencing their behavior and visibility.

4.1.1.3 Quick Fix

The Quick Fix [26] component assists developers in promptly addressing detected code issues. It collaborates with the IntelliJ Code Inspections system to identify discrepancies, inefficiencies, or potential improvements in the codebase. Upon detecting such instances, the Quick Fix component provides contextual suggestions and actions to effectively rectify the issues.

Developers can seamlessly integrate the Quick Fix component into refactoring plugins by utilizing IntelliJ IDEA's API and infrastructure. Custom Quick Fix actions can be easily defined by extending abstract classes within the IntelliJ IDEA SDK. These objects encapsulate the logic for automatically resolving detected issues or guiding users through manual correction processes.

Developers confidently utilize the IntelliJ IDEA PSI (Program Structure Interface) API to efficiently identify and resolve code issues through Quick Fix

actions. This powerful API grants access to the code's abstract syntax tree, enabling the precise identification of specific code elements that require correction. To implement custom inspections, developers can extend the LocalInspectionTool class and override its `buildVisitor()` method with confidence. This method defines the logic to traverse the project PSI and identify problematic code patterns. By utilizing this approach, developers can confidently identify and address problematic code patterns in their projects.

When a problem is detected, developers should register it with the IntelliJ Code Inspections system using the ProblemsHolder class. The ProblemsHolder class allows for the creation of detailed problem descriptions, including information such as the detected issue, its severity, and potential Quick Fix actions. Developers can enable users to efficiently resolve issues directly from the IntelliJ IDEA editor interface by linking Quick Fix actions to identified problems.

4.1.1.4 Element Visitor

Element Visitor is a design pattern in the IntelliJ Platform SDK. They enable developers to create custom visitor classes to traverse and analyze the codebase, performing specialized actions on different program elements.

Element Visitors are particularly useful in the development of automation refactoring plugins, where they play a pivotal role in automating code analysis. Developers create custom Element Visitors to find potential code smells. These visitors traverse the project, identify relevant code patterns, and apply targeted refactorings or optimizations if it is needed. This approach ensures precise and efficient refactoring, tailored to the specific needs of the project.

Implementing Element Visitors in refactoring plugins involves several key steps:

1. Custom visitor classes must be defined that implement the Element Visitor interface and override its methods to define the desired behavior for visiting different elements.
2. The project is traversed by utilizing the accept method provided by elements, where the properties and context of each visited element are analyzed.
3. Based on the analysis performed by the visitor, developers confidently register any identified problems in the Problem Holder and suggest Quick Fixes.

The use of Element Visitors in automation refactoring offers several benefits, including modularity, reusability, precision, and customization. Encapsulating traversal and analysis logic separately from the main plugin logic enables developers to create specialized visitors for different refactoring tasks. This approach enhances code maintainability and facilitates future plugin improvements.

4.1.2 IntelliJ Threading Layer

In the context of IntelliJ IDEA, which is inherently a multi-threaded application, the Threading Layer [28] assumes paramount importance in ensuring the responsiveness and stability of the IDE. It orchestrates the execution of various tasks, ranging from user interactions to background processes, while mitigating the risks of thread contention, deadlock, and other

concurrency-related issues. By enforcing strict concurrency models and providing abstractions for safe thread interaction, the Threading Layer fosters a robust and reliable environment for plugin development and IDE operation.

The layer leverages thread pools, thread-safe data structures, and synchronization primitives to coordinate concurrent activities within the IDE. Thread pools manage worker thread lifecycles, optimize resource utilization and task scheduling. Concurrent data structures, such as concurrent queues and locks, safely communicate and coordinate among threads, reducing the likelihood of data corruption or race conditions. The Threading Layer utilizes event dispatch mechanisms, such as the EDT (Event Dispatch Thread), to handle user interactions responsively while maintaining thread safety.

To ensure the compatibility and reliability of their extensions, plugin developers must understand and strictly adhere to the threading model enforced by the IntelliJ Threading Layer. By utilizing strong, action-oriented language and emphasizing the importance of adhering to the threading model, plugins must adhere to established guidelines and best practices for thread-safe programming. This involves avoiding long-running operations on the UI thread and using proper synchronization mechanisms when accessing shared resources. Furthermore, developers should utilize APIs provided by IntelliJ IDEA, such as `invokeLater()` and `invokeAndWait()`, to safely interact with the UI and perform background tasks asynchronously.

4.2 Plugin Configuration

IntelliJ IDEA's plugin architecture follows the principles of extensibility and modularity. This allows developers to extend the functionality of the IDE by creating custom plugins.

The `plugin.xml` configuration file is the core component of an IntelliJ IDEA plugin. It clearly defines the plugin's identity, version, description, and dependencies, providing a structured framework for seamlessly integrating new features into the IDE environment. Example of `plugin.xml` structure is shown on Figure 4.3.

```
<idea-plugin>
    <id>com.example.plugin</id>
    <name>Plugin</name>
    <description>A description of a plugin</description>
    <version>1.0</version>

    <actions><!-- actions --></action>

    <extensions defaultExtensionNs="com.intellij">
      <inspectionToolProvider <!--inspections provider-->/>
      <projectConfigurable <!--plugin settings-->/>
    </extensions>
</idea-plugin>
```

Figure 4.3 – Example of plugin configuration file

Components of `plugin.xml` include:

1. *Plugin Identification*:

a. `<id>`. Unique identifier for the plugin, typically in reverse domain name notation.

b. `<name>`. Human-readable name of the plugin.

c. `<description>`. Brief overview or summary of the plugin's purpose and functionality.

d. `<version>`. Version number of the plugin.

2. Extensions. Extensions delineate the additional functionalities or integrations provided by the plugin. These can range from custom actions and code inspections to tool windows and language support. Extensions are declared within the `<extensions>` tag, each specifying a particular extension point and associated implementation.

a. *Actions.* Actions represent user-triggered operations within IntelliJ IDEA, such as menu items, toolbar buttons, or keyboard shortcuts. Action declarations reside within the `<actions>` tag (Figure 4.4), specifying the action ID, class, text, and description.

```
<actions>
    <action
      id="Action"
      class="com.example.ActionClass"
      text="Action"
      description="Description of an action"/>
</actions>
```

Figure 4.4 – Example of actions section

b. *Settings.* Settings extensions facilitate the integration of custom settings pages into IntelliJ IDEA, enabling users to configure plugin behavior according to their preferences. Settings configurables are declared within the `<applicationConfigurable>` tag, specifying the configurable class and display name (Figure 4.5).

```
<applicationConfigurable
      displayName="Plugin Settings"
      configurableClass="com.example.SettingsConfigurable"/>
```

Figure 4.5 – Example of settings section

c. *Code Inspections.* Code inspections analyze the source code for potential issues, errors, or improvements, enhancing code quality and maintainability. Inspection providers are declared within the `<localInspectionTools>` tag, specifying the inspection provider class, which include all implmented indpections (Figure 4.6).

```
<localInspectionTools>
            <inspectionTool
            class="com.example.InspectionProviderClass"/>
</localInspectionTools>
```

Figure 4.6 – Example of code inspections section

IntelliJ IDEA's plugin architecture simplifies development by allowing developers to focus on individual components, like actions and settings, separately. With the convenience of the `plugin.xml` configuration file, developers can avoid building complex structures and efficiently enhance the IDE's functionality to suit their needs.

4.3 Code Inspections Design

The plugin's code inspections subsystem is primarily divided into two fundamental components:

1. *Detection Mechanism.* This component is responsible for conducting code inspections and analyses to identify code smells within the source code.
2. *Correction Mechanism.* This component aids developers in the refactoring process to address the identified code smells effectively.

Each of these components is further subdivided into various modules, as depicted in Figure 4.7. The subsequent sections will delve into each component and its constituent modules in greater detail.

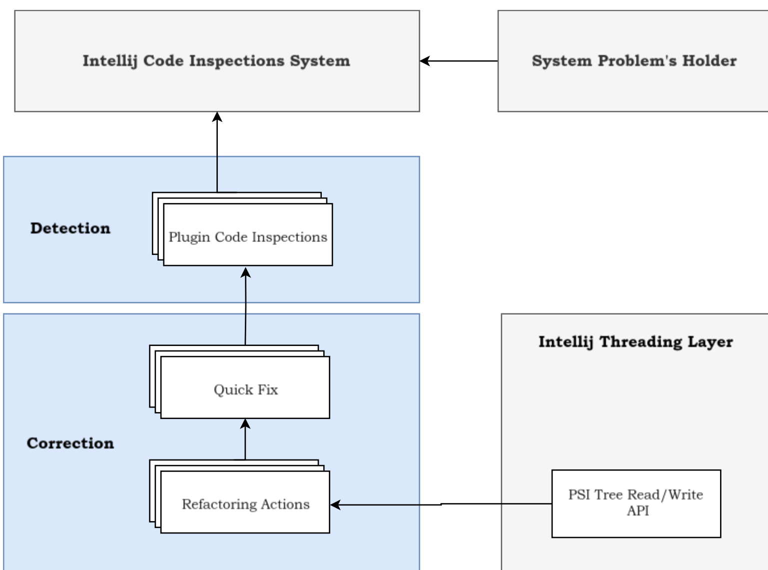


Figure 4.7 – Plugin inspections diagram

4.3.1 Detection Mechanism

To implement a custom code inspection, it is needed to create a new class extending IntelliJ `AbstractBaseJavaLocalInspectionTool`. This superclass provides essential functionalities and interfaces for defining code inspections.

Upon nesting the custom inspection, developers override key methods to define inspection logic and behaviors. These overridden methods play a pivotal role in facilitating code analysis and issue detection within the IDE. Notable methods commonly overridden include:

1. `buildVisitor(holder: ProblemsHolder, isOnTheFly: Boolean)`. This method is overridden to specify the visitor pattern implementation for traversing the code's PSI (Program Structure Interface) elements. Within this method, developers define the logic for analyzing PSI elements and detecting code issues based on predefined rules.
2. `getDisplayName(): String`. Overriding this method allows developers to specify the display name for the custom code inspection within IntelliJ IDEA's inspection settings. Providing a descriptive and intuitive display name enhances usability and clarity for users configuring inspection preferences.

3. `getShortName(): String`. This method is overridden to define a unique identifier or short name for the custom inspection. The short name serves as a reference point for identifying the inspection programmatically and distinguishing it from other inspections within the plugin.

Within the overridden `buildVisitor()` method, developers implement custom inspection logic tailored to project-specific requirements. Leveraging IntelliJ IDEA's PSI traversal capabilities, developers traverse the code's abstract syntax tree (AST) to analyze code elements and identify potential issues. By incorporating predefined rules and coding standards, developers can systematically detect and highlight code quality issues, empowering users to address them proactively. Class diagram shown on Figure 4.8.

The plugin's detection mechanism is based on the IntelliJ Inspections System, which provides a range of functions for developing custom code inspections. This connection not only speeds up the entire plugin development process, but it also provides a consistent user experience by aligning with other built-in inspections.

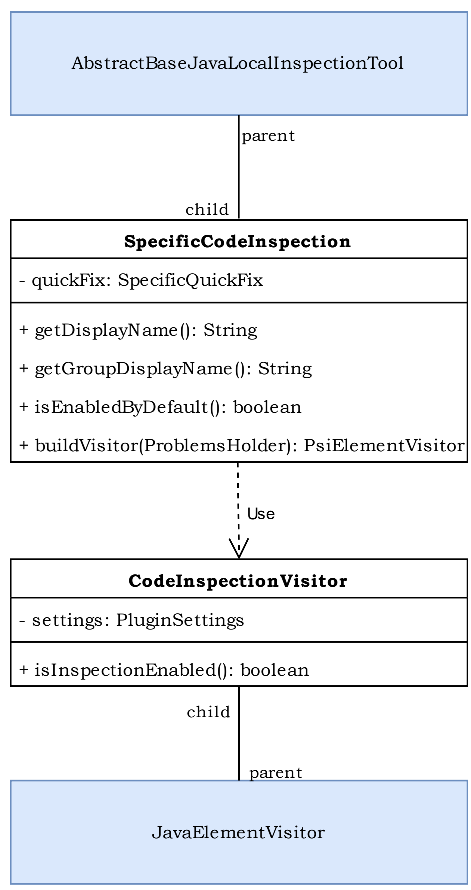


Figure 4.8 – Detection mechanism class diagram

4.3.2 Correction Mechanism

To embark on crafting a custom Quick Fix, developers should devise a new class that adheres to the `LocalQuickFix` interface. This interface lays the groundwork for specifying corrective actions that can be seamlessly integrated directly from the code editor, offering a streamlined solution to coding obstacles.

The essence of tailoring a custom Quick Fix lies in the strategic overriding of key methods subsequent to its integration within LocalQuickFix. The

methods targeted for overriding are used for instigating code modifications and addressing the identified issues efficiently. Among these, the most commonly overridden methods include:

1. `applyFix(project: Project, descriptor: ProblemDescriptor).` A pivotal method where developers delineate the corrective actions. This involves detailing the code adjustments or transformations necessary to ameliorate the identified problem.
2. `getName().` This method, when overridden, furnishes a descriptive name for the Quick Fix, enhancing its visibility and accessibility in the IDE's user interface, thereby aiding users in pinpointing the precise Quick Fix required.
3. `getFamilyName().` Overriding this method enables the categorization of Quick Fixes into coherent groups or families, based on their utility. This organization streamlines the selection process for users within the IDE's Quick Fix menu.

Also the `applyFix()` method can customized to address specific code issues using IntelliJ IDEA's Program Structure Interface (PSI) for code inspection. By navigating the abstract syntax tree (AST), developers can easily identify and correct code elements, ensuring issue resolution. Class diagram shown on Figure 4.9.

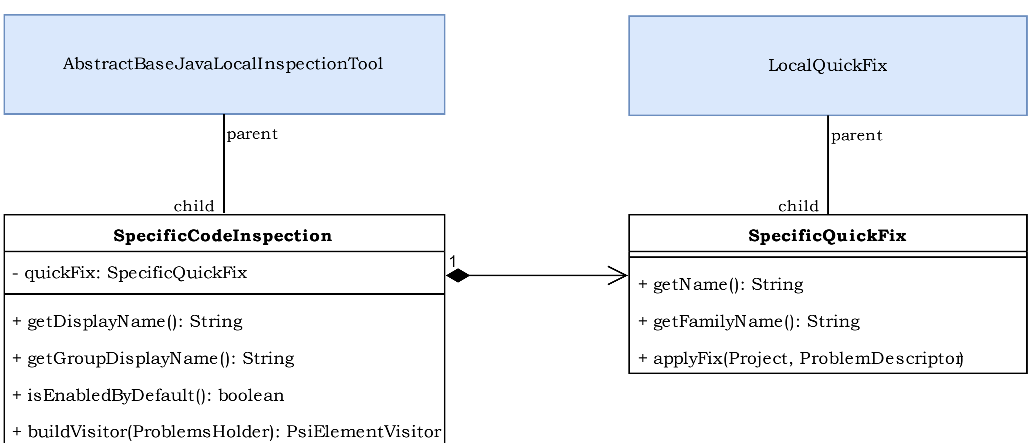


Figure 4.9 – Correction mechanism class diagram

An advanced feature of IntelliJ IDEA is the ability to implement refactoring in phases through refactoring dialogues, allowing for a more nuanced approach to modifying code. Developers are provided with a range of solutions to improve code quality and readability, including proposing alternative method names. Figure 4.10 illustrates this iterative refactoring process, depicting a series of decision points following each dialogue that ensure the developer's agreement before proceeding. It emphasizes the condition-based progression of the refactoring tasks, ensuring that each phase is completed satisfactorily before the workflow continues to the subsequent steps.

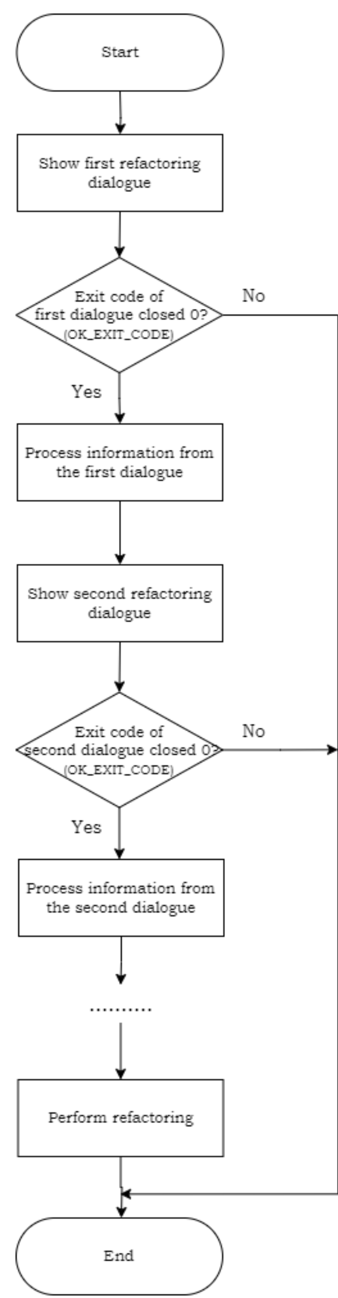


Figure 4.10 – Example of a refactoring workflow using dialogues

4.4 User Interface Design

Before delving into the specifics of UI design, it's important to note that code inspection does not require any additional UI development. IntelliJ IDEA already provides a robust code inspection mechanism that highlights problems and suggests fixes to the user. Therefore, the already implemented mechanism would provide additional code inspections for users.

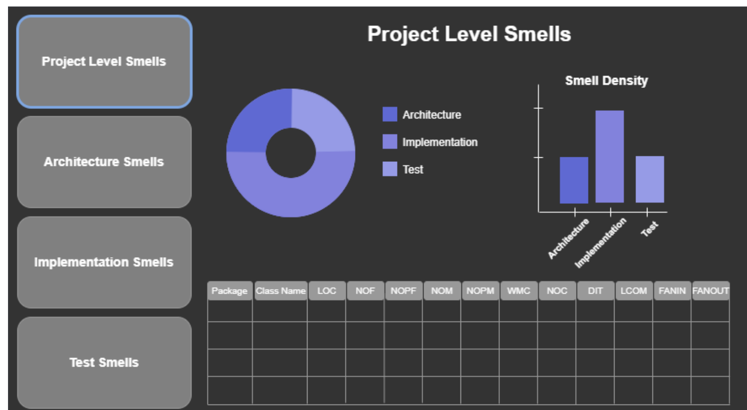
The primary requirement for our UI design is the implementation of a project analysis panel. The project analysis functionality within a refactoring plugin would be placed in the top menu under "Tools". This placement ensures accessibility and visibility, allowing developers to easily access project analysis

tools without cluttering the main interface. The panel will act as a central hub for displaying key project analytics, including:

1. *Code smells by category*. Users should be able to quickly assess the health of their codebase by viewing the number of code smells categorised by type.
2. *OOP metrics*. The Project Analysis Panel should provide insight into the object-oriented metrics of the project, as described in Section 3.2.2.

Two themes have been developed to visualise the project analysis panel - one for dark and one for light IntelliJ IDEA themes. Each theme includes selected colour palettes for graphs to ensure clarity and consistency in data presentation. The design models serve as orientation points for the UI implementation, aiming to capture the overall look and feel without the need for pixel-perfect replication.

The dark theme design (Figure 4.11) uses a combination of dark grey and vibrant accent colours to maintain readability and visual appeal. Charts within the Project Analysis panel are presented against a dark background to reduce eye strain and effectively highlight important data points.



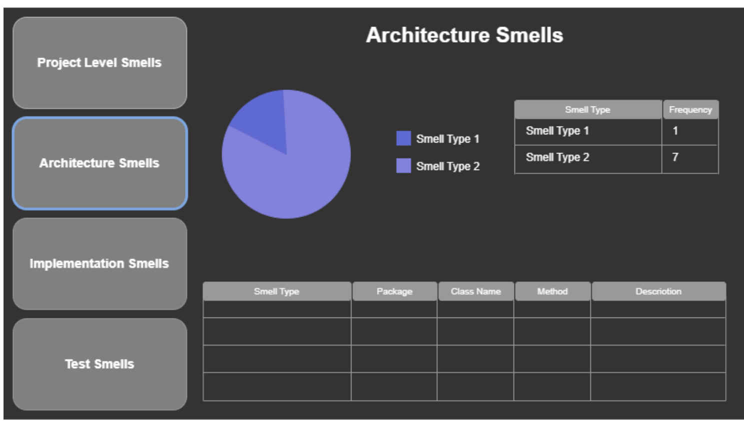


Figure 4.11 – Dark theme “Project analysis” panel design

Conversely, the light theme design (Figure 4.12) uses a brighter colour scheme with subtle contrasts to improve readability in well-lit environments. Soft pastel colours are used for charts to maintain a cohesive visual aesthetic while providing clear distinctions between data categories.

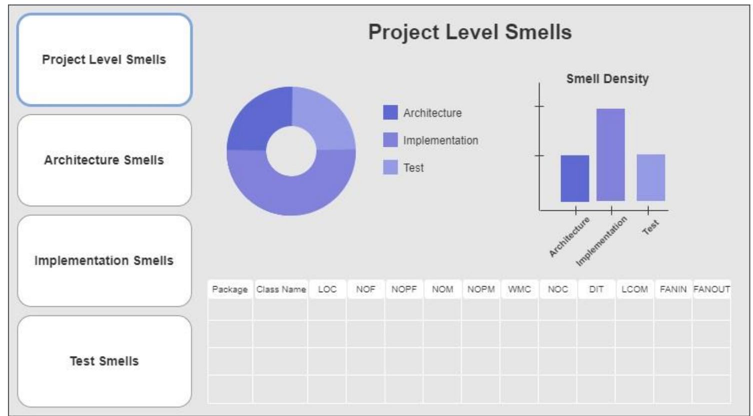


Figure 4.12 – Light theme “Project analysis” panel design

Designing the UI for an Idea refactoring plugin involves striking a balance between functionality and aesthetics. By leveraging IntelliJ IDEA's standard code inspection mechanism and focusing on improving the project analysis panel, we can provide users with valuable insight into their codebase while maintaining a seamless user experience. The dark and light theme designs, along with carefully selected colour palettes, ensure that users can comfortably analyse their projects across different environments and preferences.

4.5 Customization Menu Design

To ensure a user-friendly and easily accessible customization interface, a custom submenu named “AutoRefactor Plugin Settings” will be created within the “Settings/Tools” section of the IntelliJ IDEA. This placement is chosen as it is associated with the existing tools and settings used by developers for code manipulation and environment configuration. The “Settings/Tools” section is the central hub for modification and adjustment tools, making it an intuitive location for users seeking to customize their refactoring operations (Figure 4.13). This strategic placement enhances discoverability and aligns with the user's workflow, promoting a seamless integration of the plugin into the daily activities of the developer.

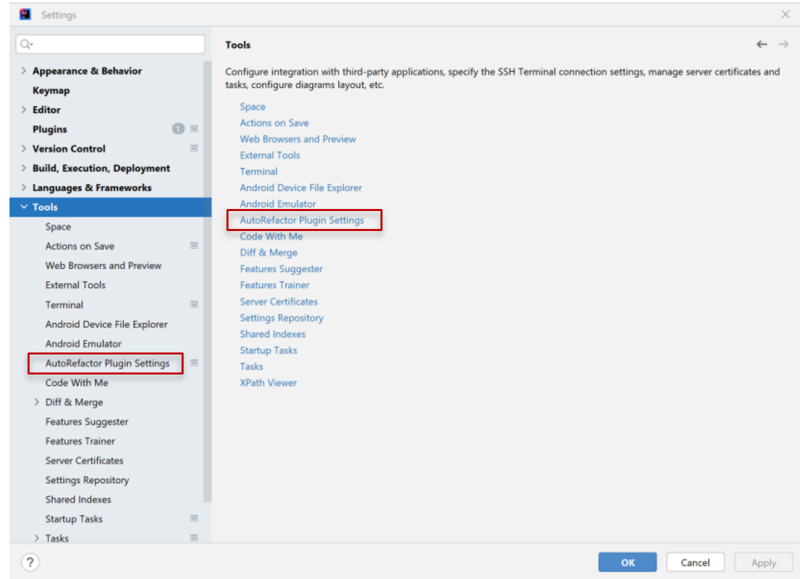


Figure 4.13 – Settings dialogue

The customization dialogue offers several options to customize the refactoring process. One of the most important options is the ability to set a maximum allowed cyclomatic complexity for methods, with an upper limit of 20. This limit is advocated by McCabe's seminal work on cyclomatic complexity, which suggests that a higher cyclomatic complexity in methods correlates with increased risk of errors, making the code more challenging to test, maintain, and understand [29]. Also, the dialog contains checkboxes for each available inspection, allowing users to enable or disable specific refactoring inspections based on their project requirements or personal preferences. Some inspections require user input, such as defining a pattern for test methods naming convention or setting a maximum allowed cyclomatic complexity.

The test naming convention section allows developers to define a regular expression (regex) pattern that test method names must conform to in order to ensure consistency and readability across the codebase. This field accepts a string and after pressing "*OK*" or "*Apply*" buttons performs a validation of the regular expression by calling `Pattern.compile`. If the string can be compiled as a Pattern, then it is a valid pattern for test conventions, otherwise the user will see an error message just below the text field.

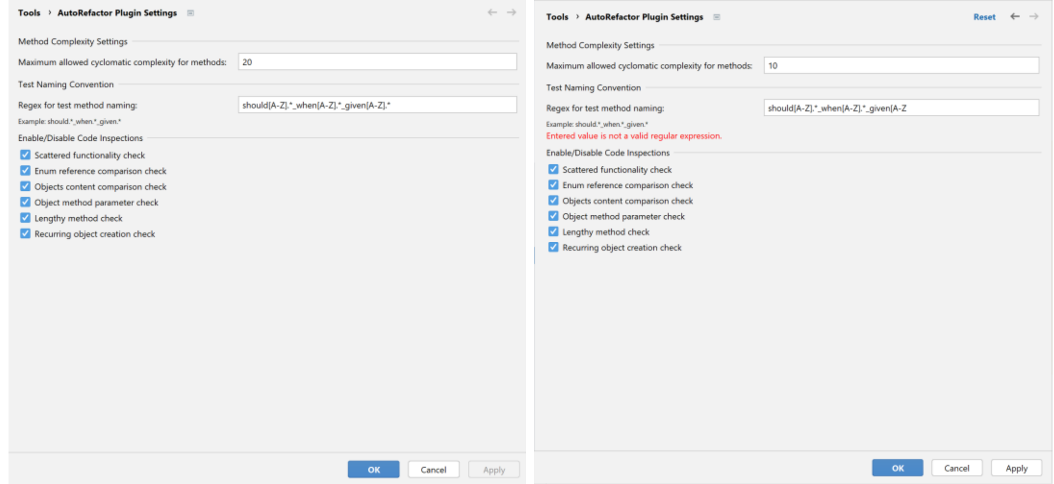


Figure 4.14 – Plugin settings

This level of customization ensures that developers can fine-tune the plugin's functionality to align with their coding standards and practices, thereby enhancing code quality and developer efficiency. Settings dialogue for plugin shown on Figure 4.14.

The architecture for the IntelliJ IDEA plugin settings is expertly designed to provide a seamless and efficient user experience for configuring the plugin. As shown in Figure 4.15, the primary components involved in the settings management are interrelated and work together seamlessly.

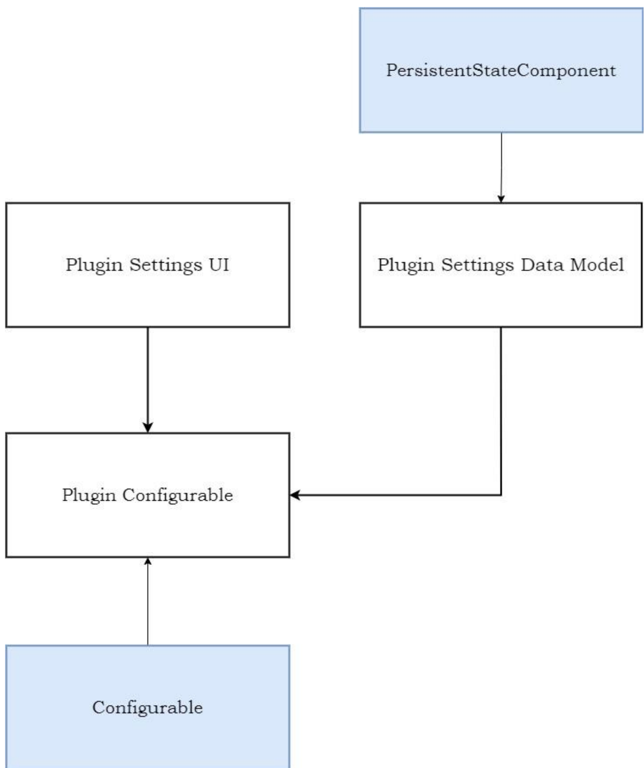


Figure 4.15 – Settings components diagram

The "Plugin Settings Data Model", serves as the foundational data model within the plugin settings architecture. This component encapsulates the state of user-configurable settings, maintaining their integrity and persistence throughout the plugin's lifecycle. Each setting parameter is stored as a strongly typed property, which enhances both: type safety and encapsulation. To ensure the persistence of these settings, the component implements the `PersistentStateComponent` interface. This interface is crucial as it defines methods for loading and saving the configurable state, thus embedding persistence directly into the data model. The component promotes high cohesion within the system architecture by managing only the settings data and its integrity.

The "Plugin Settings UI" component serves as the user interface representation of the data model within the plugin architecture. It is designed to reflect and interact with the underlying data model, allowing users to effectively view and modify settings. This component consists of various UI elements such as text fields, labels, checkboxes and other interactive controls, each of which is linked through "Plugin Configurable" to corresponding settings in the data model. These elements are organised into a cohesive panel, providing an intuitive user experience while maintaining a clear visual hierarchy.

The “Plugin Configurable” component implements the `Configurable` interface which defines the methods required to construct a user configuration interface within the plugin. This interface separates the configuration UI from the specifics of the settings management. It establishes a clear dependency relationship with the data model, emphasizing the interface's reliance on the component for accessing and manipulating the plugin settings. This relationship improves modularity within the plugin architecture and enables clear separation of concerns.

4.6 Algorithmic Design of Components

4.6.1 OOP Metrics Calculation

This chapter expands on the basic requirements set out in Section 3.2.1 by detailing the development of a metrics calculation plugin for the IntelliJ IDEA environment. The plugin, which uses the IDE's Program Structure Interface (PSI), is designed to calculate critical metrics that measure the quality, complexity, and maintainability of software.

4.6.1.1 Lines of Code (LOC)

The LOC metric acts as a preliminary indicator of the size and potential complexity of a class by quantifying its total number of lines. This measure is derived by directly examining the content of the class by calculating the number of line terminators present within the class definition. An extra line is counted if the class definition does not end with a newline character.

4.6.1.2 Number of Fields (NOF) for classes

NOF evaluates the structural complexity and the extent of data encapsulation within a class by counting its fields. This metric is computed by extracting and measuring the length of the `PsiField[]` array from the `PsiClass`.

4.6.1.3 Number of Public Fields (NOPF) for classes

NOPF assesses the class’s approach to data encapsulation by identifying the number of fields that are publicly accessible. The count is performed by iterating over the `PsiField` array and checking each field for the `PsiModifier.PUBLIC` modifier.

4.6.1.4 Number of Methods (NOM) for classes

NOM quantifies the functional complexity of a class by counting its methods. It leverages the `PsiMethod[]` array from the class, tallying its length reflecting the class's behavioral capabilities.

4.6.1.5 Number of Public Methods (NOPM) for classes

NOPM assesses the complexity of a class's external interface by counting the methods that are publicly accessible. The algorithm operates by iterating over each method in the class, incrementing a counter whenever a method is identified with the `PsiModifier.PUBLIC` attribute, thus providing a direct measure of the class's accessibility and openness.

4.6.1.6 Weighted Methods per Class (WMC) for classes

The algorithm estimates the Weight of Class (WMC) metric, which provides insight into the complexity of a class by evaluating its methods. However, the current approach only counts the methods directly declared in the class, including inherited methods, but excluding constructors, static initialisation blocks and methods inherited from the `Object` class.

4.6.1.7 Number of Children (NC) for classes

NC measures the number of direct subclasses extending from the class, illustrating its polymorphic impact within the inheritance hierarchy. This metric is obtained by using `ClassInheritorsSearch.search(psiClass, scope, true)` to identify subclasses.

4.6.1.8 Depth of Inheritance Tree (DIT) for classes

DIT calculates the depth of inheritance from the class to the root class, providing a measure of design complexity. The depth is determined by tracing the inheritance chain using `PsiClass.getSuperClass()` until no more superclasses are found, excluding the class itself from the final count.

4.6.1.9 Lack of Cohesion in Methods (LCOM) for classes

LCOM evaluates the cohesion between methods based on their shared field accesses, highlighting potential areas for redesign. This metric evaluates each possible method pair for shared fields via `methodsShareField(PsiMethod method1, PsiMethod method2)`, counts these interactions, and calculates LCOM as the difference between the total number of method pairs and those that share fields, algorithm is presented on Figure 4.16.

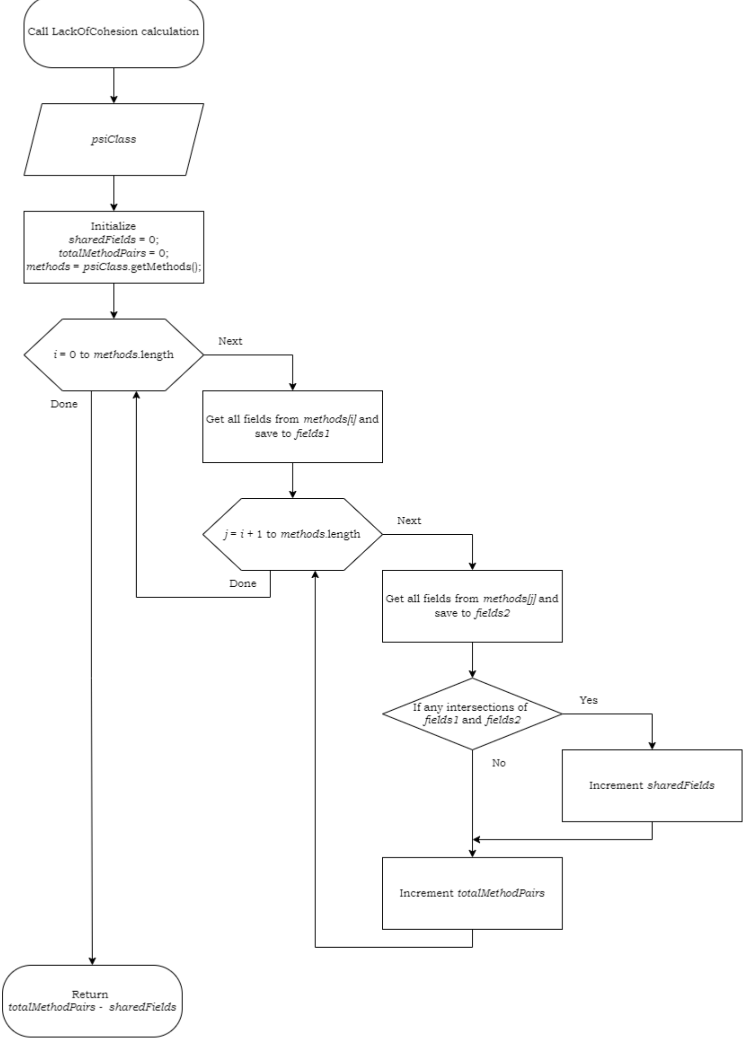


Figure 4.16 – Lack of cohesion for class calculation algorithm

4.6.1.10 Fan-in (FANIN) and Fan-out (FANOUT) for classes

FANIN and FANOUT determine the degree of coupling within the project by counting the number of classes that either use the given class (FANIN) or are dependencies of the class (FANOUT). These metrics are calculated by identifying and counting the unique classes that reference or are referenced by the methods of the class. Algorithms for calculating these metrics are shown in Figure 4.17.

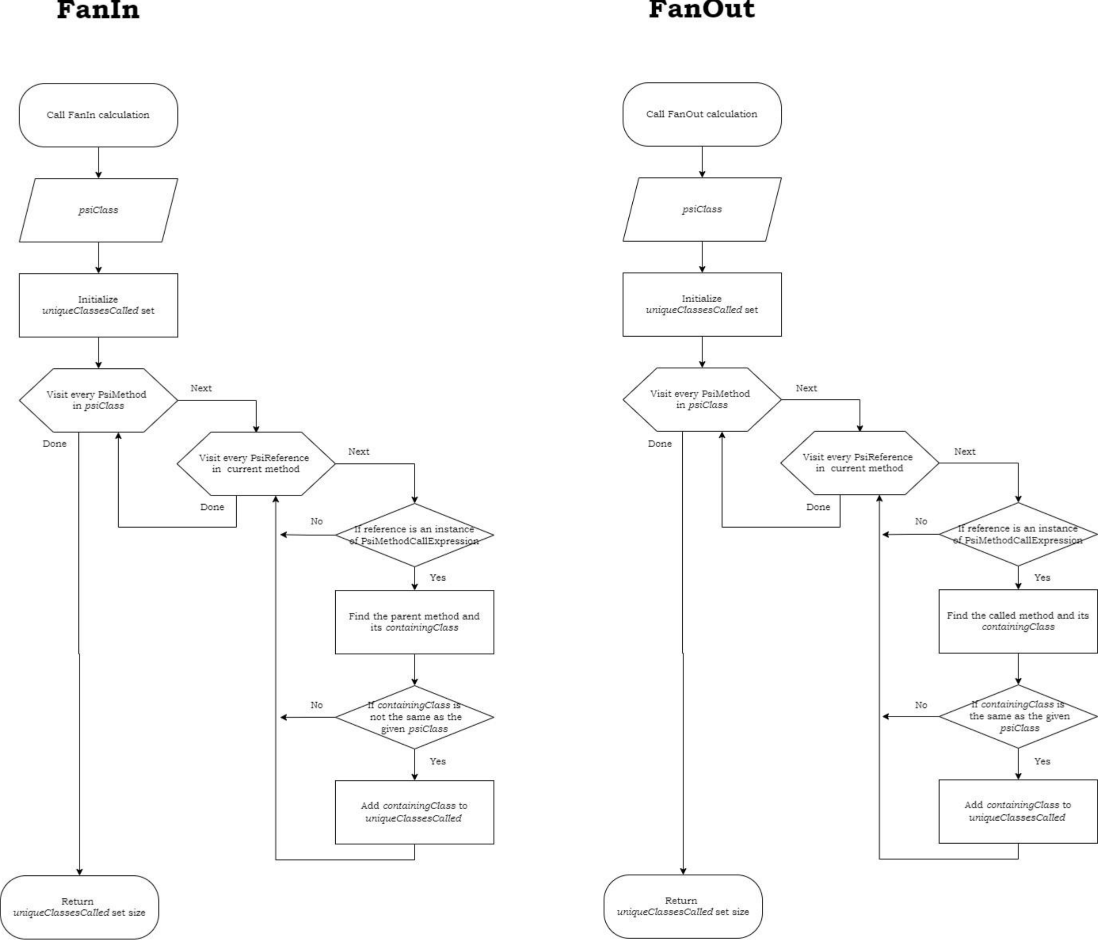


Figure 4.17 –FanIn and FanOut calculation algorithms

4.6.2 Code Smells Detection and Refactoring

Chapter 3.2.2 articulated the essential requirements for the effective implementation of code inspections within the design framework. This chapter critically examines the methodologies and strategies that address these requirements, emphasising their academic and practical importance. The discussion highlights the procedural aspects of conducting comprehensive code inspections, while emphasising the key role that these inspections play in improving software quality and reliability. Adherence to the principles outlined can significantly reduce potential design defects, thereby increasing the robustness and efficiency of software systems.

4.6.2.1 Architectural Code Smells

4.6.2.1.1 Scattered Functionality Identification

4.6.2.1.1.1 Algorithm

Scattered functionality occurs when related functionality is duplicated across multiple locations within a codebase. This code smell indicates a lack of cohesion in the code, making it difficult to maintain and understand. The goal of the designed algorithm is to identify scattered functionality to improve code cohesion and quality.

The algorithm shown on Figure 4.18 provides a structured approach to identifying scattered functionality in code. The process follows three main phases: checking that inspection is enabled, normalising each block of code, and checking for repeated elements within the map.

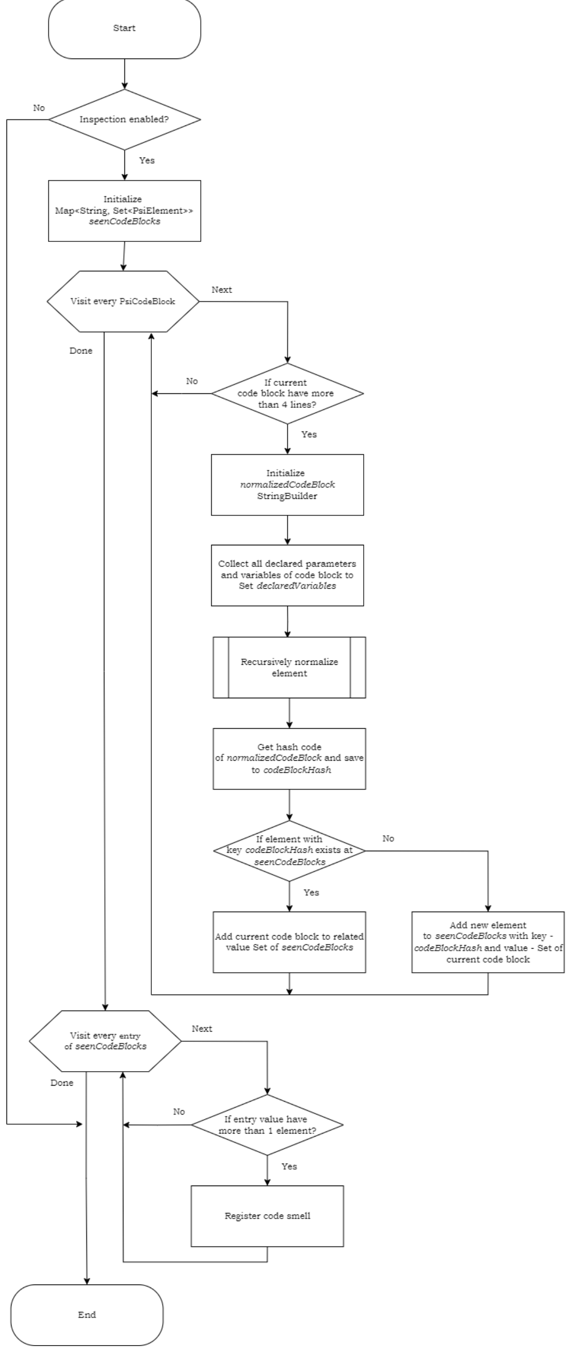


Figure 4.18 – Scattered functionality detection algorithm

The algorithm starts by determining whether inspection is enabled. If so, the algorithm initialises a map for storing instances and normalised elements.

The algorithm then proceeds to normalise each block of code (`PsiCodeBlock`). A `PsiCodeBlock` specifically refers to a structured segment of code within braces {}, defining a logical scope in programming. During this phase, it iteratively inspects each block of code, focusing on removing variable names to prevent scattered functionality. After normalising the elements, the algorithm stores them in a map where the key is the hash code of the normalised element, and the value is a set of code blocks. More detailed algorithm of recursive normalisation shown on Figure 4.19.

In the final stage, the algorithm checks whether any value from the map contains more than one repeated code block. This process helps to identify instances of scattered functionality, where similar or related code blocks are scattered in different locations. If such instances are found, the algorithm registers a code smell, indicating the need for refactoring.

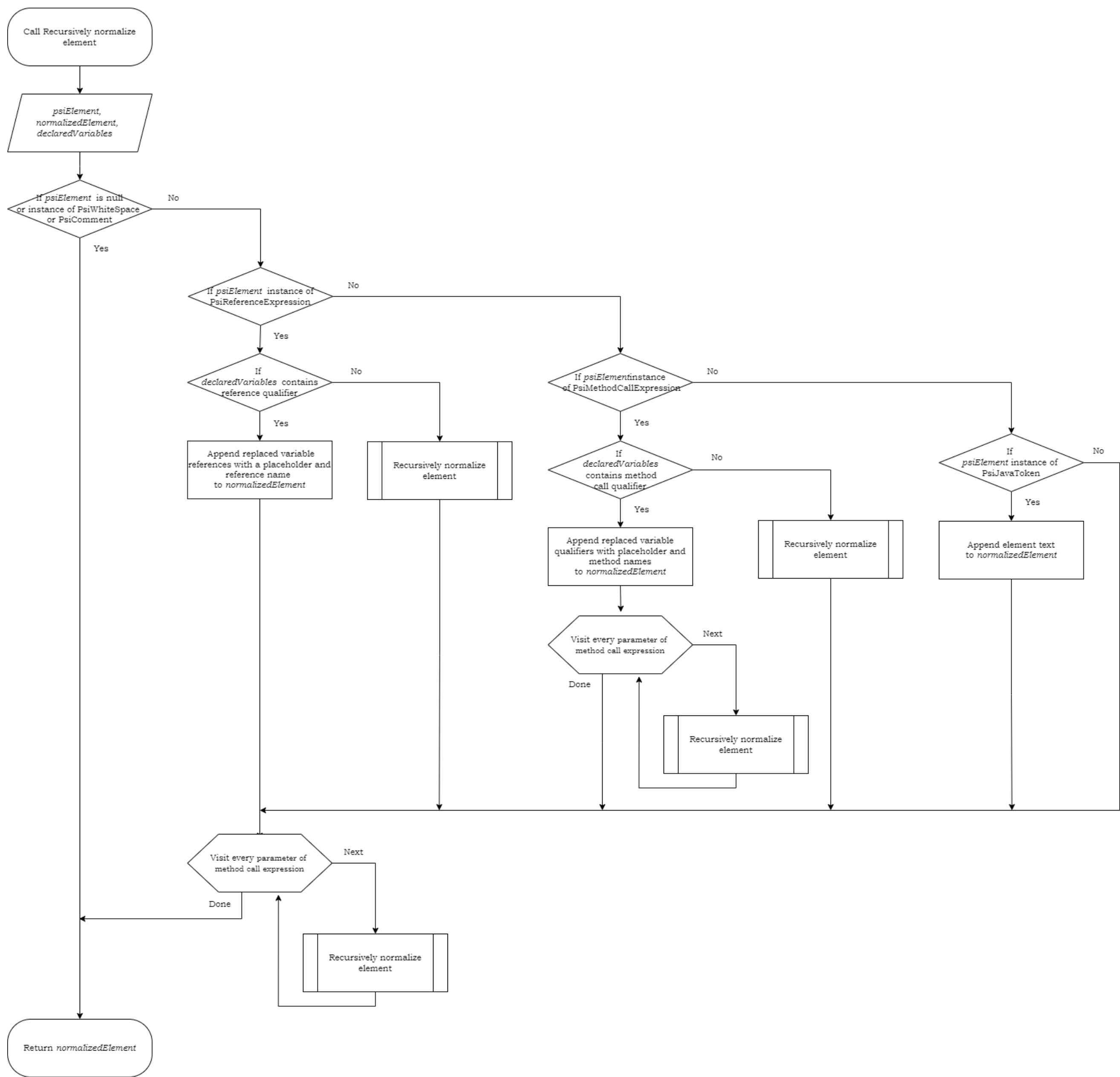


Figure 4.19 – Recursive Psi element normalization algorithm

In developing the algorithm, it was decided not to implement a quick fix for several key reasons. While quick fixes are convenient, they often address the symptoms rather than the underlying problems, which can lead to a variety of problems in the long run.

Firstly, quick fixes can mask deeper design flaws within the codebase. Scattered functionality typically indicates a lack of cohesion or poor architectural structure. Implementing a quick fix would likely only provide a superficial solution, leaving the underlying problem unresolved. This could lead to further complications in the future as the underlying issues remain unaddressed.

Second, quick fixes for architectural problems can compromise code quality. Effective maintenance and evolution require clean, well-structured code that adheres to best practices.

4.6.2.1.1.2 Limitations

The algorithm presented in the previous chapter provides an effective means of detecting and normalising scattered functionality in code. However, it is important to recognise its limitations in order to further improve its performance and effectiveness. The main limitation of the algorithm lies in its focus on analysing blocks of code as discrete entities. While this approach is useful for identifying dispersed functionality at a broader structural level, it does not consider instances where related functionality is dispersed, for example within a single code block.

To address this limitation, future versions of the algorithm should consider analysing different combinations of lines of code within a single code block. The following strategies could be used:

1. *Fine-Grained analysis*. The algorithm could be enhanced to perform a fine-grained analysis of code lines within each block. By examining sequences of code lines and their logical relationships, the algorithm could identify and flag potential cases of scattered functionality that occur within individual blocks.
2. *Sliding window technique*. Implementing a sliding window technique could help the algorithm identify related code lines within a block. This technique would involve examining subsets of code lines using a moving window of fixed or variable size, allowing the algorithm to detect patterns or sequences indicative of scattered functionality. But this algorithm can be resource consuming.
3. *Graph-based analysis*. A graph-based analysis approach could be used to model the relationships between code lines within a block. By representing the code as a graph and analyzing the connections and dependencies, the algorithm could uncover hidden cases of scattered functionality that linear analysis might miss.

The limitations highlighted above can be addressed in future versions of the plugin. By implementing the suggested enhancements, the algorithm can be further refined to provide more comprehensive detection of scattered functionality, both within and across code blocks.

These improvements would contribute to the development of robust and efficient software systems, as the enhanced algorithm would be better equipped to identify and refactor problematic code structures. Future versions of the plugin could incorporate these changes to deliver enhanced functionality and improved software quality.

4.6.2.2 Implementation Code Smells

4.6.2.2.1 Reference Object Comparison Instead of Content Comparison Detection

Comparing objects with == or != in Java is a code smell that can lead to incorrect behaviour and bugs. The reason for this is that these operators check for equality of references rather than equality of values. Reference equality checks only whether two references point to the same object in memory, which is often not the intended comparison for objects.

In contrast, the `Objects.equals` and `equals` methods compare the actual contents of the objects. However, the use of `Objects.equals` is generally better than the direct use of the `equals` method, because it handles null cases. If one of the objects being compared is `null`, `Objects.equals` returns `false`, unless both are `null`, in which case it returns `true`. This is a common edge case that `Objects.equals` handles cleanly, whereas the `equals` method can throw a `NullPointerException` if the left operand is `null`.

The inspection algorithm depicted in the Figure 4.20 begins by checking if inspection is enabled. If it is, the algorithm proceeds to visit each `PsiBinaryExpression` in the codebase. For each expression, it checks if the operation type is == or !=. If this condition is met, it further checks if both operands are objects and not enums. If both conditions are satisfied, the algorithm registers a code smell, indicating the need of refactoring.

Code Inspection **Quick Fix**

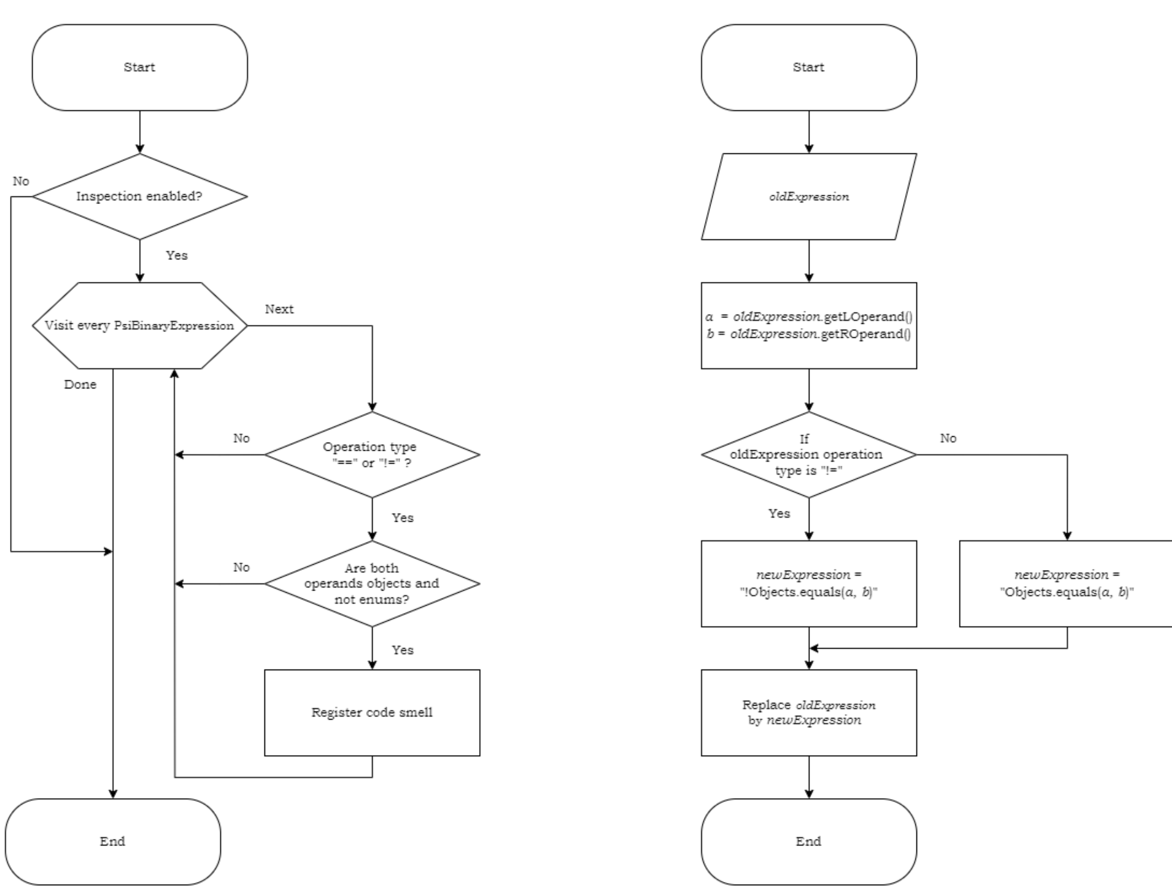


Figure 4.20 – Object reference comparison code inspection and quick fix algorithms

The quick fix algorithm, presented on the same figure as inspection algorithm, starts with an expression to fix. The algorithm extracts the left (`a`) and right (`b`) operands of the expression. It then checks if the operation type is reference comparison. If the operation type is ==, the algorithm creates a new expression - `Objects.equals(a, b)`. If the operation type is !=, the algorithm creates a new expression - `!Objects.equals(a, b)`. The algorithm then replaces the old expression with the new expression, thereby concluding the quick fix process.

4.6.2.2.2 Content Enums Comparison Instead of Reference Comparison Detection

Comparing enums in Java using the equals method is a code smell that can lead to inefficient or incorrect behaviour. Enums are best compared using the == operator, which evaluates reference equality. Because each enum instance is different, == provides a clearer and more efficient approach to comparing enums, unlike the equals method, which is typically used to compare object contents.

The inspection algorithm, shown in the flowchart on Figure 4.21, starts by checking whether inspection is enabled. If it is, the algorithm proceeds to visit each `PsiMethodCallExpression` in the codebase. For each expression, it checks whether the method is equal and whether both operands are enums. If both conditions are met, the algorithm registers a code smell, indicating the need for refactoring.

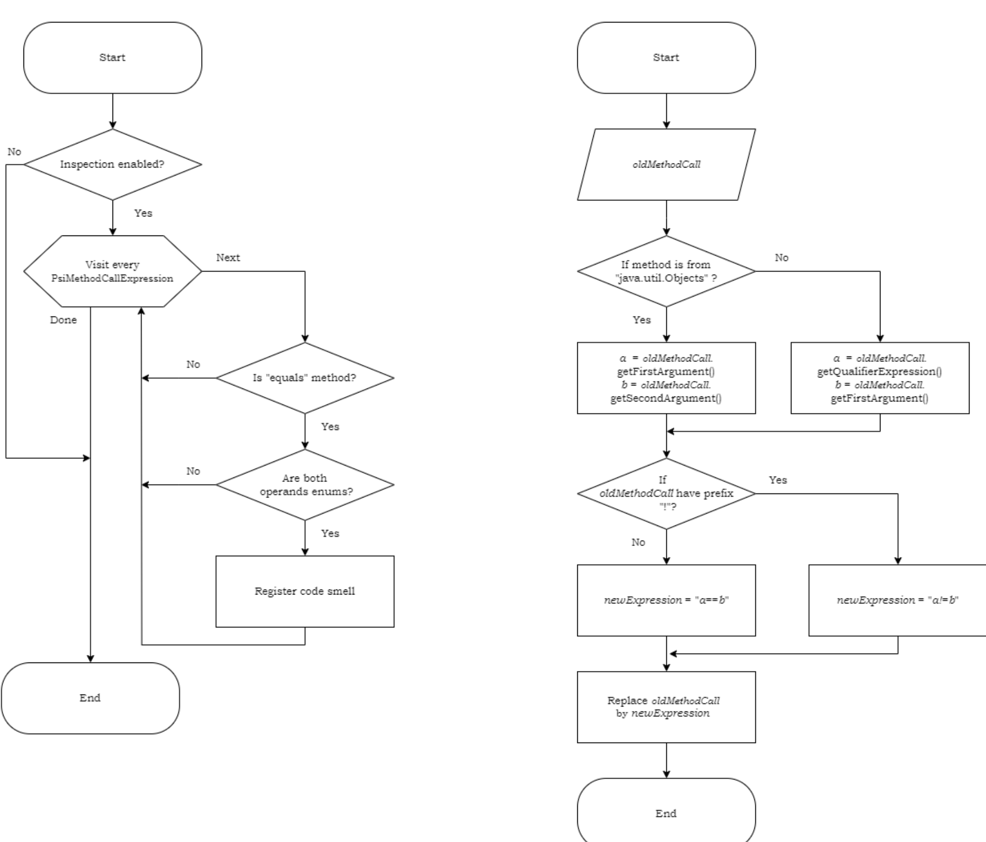


Figure 4.21 – Enums content comparison code inspection and quick fix algorithms

The quick fix algorithm, shown on the same figure, initiates with receiving the old method call that needs correction. It first determines if the method is part of the `java.util.Objects` class. If so, it retrieves the arguments used in the method call, typically the operands being compared. If the old method call is determined to have the negation operator "`!`" as a prefix, indicating a negated comparison, the algorithm will generate the new expression - `a!=b`. If there is no negation, the new expression is set to `a==b`. This approach ensures that enums are compared using the most appropriate Java operator, promoting both performance optimization and code readability. The oldMethodCall in the source code is then replaced with this newExpression, completing the quick fix process. This replacement corrects the identified code smell by using the correct operator for enum comparison, aligning with best practices in Java programming.

4.6.2.2.3 Object Method Parameters Check

The current code smell relates to methods that accept an object as a parameter and then use or access only a single property of that object. This type of code smell is problematic because it indicates poor use of the object's potential and leads to several problems:

1. *Wasted memory and resources*. Passing an entire object when only one of its properties is used is inefficient. The overhead of creating and managing the object is wasted when a simpler data type would suffice.
2. *Loss of clarity*. If a method takes an object as a parameter but uses only a single property, this can be misleading to anyone reading the code. It implies that the method may need or use more of the object's properties, which can lead to confusion and incorrect assumptions.
3. *Reduced reusability*. Methods that depend on specific objects when they only need a single piece of data are less reusable. This reduces the flexibility of the code and makes it harder to adapt or extend functionality in the future.
4. *Poor abstraction*. Good abstraction in code allows for clear and intuitive interfaces. Using an entire object where a single value would suffice indicates that the abstraction is not well thought out, which can make it difficult to maintain and understand.

Code inspection and quick fix algorithms are shown in Figure 4.22. The identification process for this code smell starts by examining methods that accept objects as parameters. It checks whether these methods access only one property of the object passed. This is done by iterating through each method and inspecting each parameter to see if it is an object. The method then records each property of the object that is accessed. If, at the end of the inspection of the method, it turns out that only one property of the object is used, this is flagged as a code smell. This suggests that the method does not need the whole object, but only certain pieces of information from it, indicating an overuse of object passing that could be streamlined.

The quick fix is to minimise unnecessary dependencies by changing the method parameters to include only the necessary data. Instead of the whole object, the method parameters are refactored to accept only the properties that are actually used within the method. This involves changing the method signatures to replace object parameters with parameters representing only the required properties. This refactoring reduces coupling and improves the cohesion of the method, making it clearer and more focused on its actual requirements. In addition, it can improve performance by reducing the overhead associated with passing large objects, especially when only a small portion of the object is needed.

This approach to identifying and fixing code smells not only simplifies the method interface, but also encourages better design practices, promoting the creation of methods that are more modular and easier to understand, test and maintain. By focusing on the actual data needs of each method, the entire codebase becomes more efficient and robust.

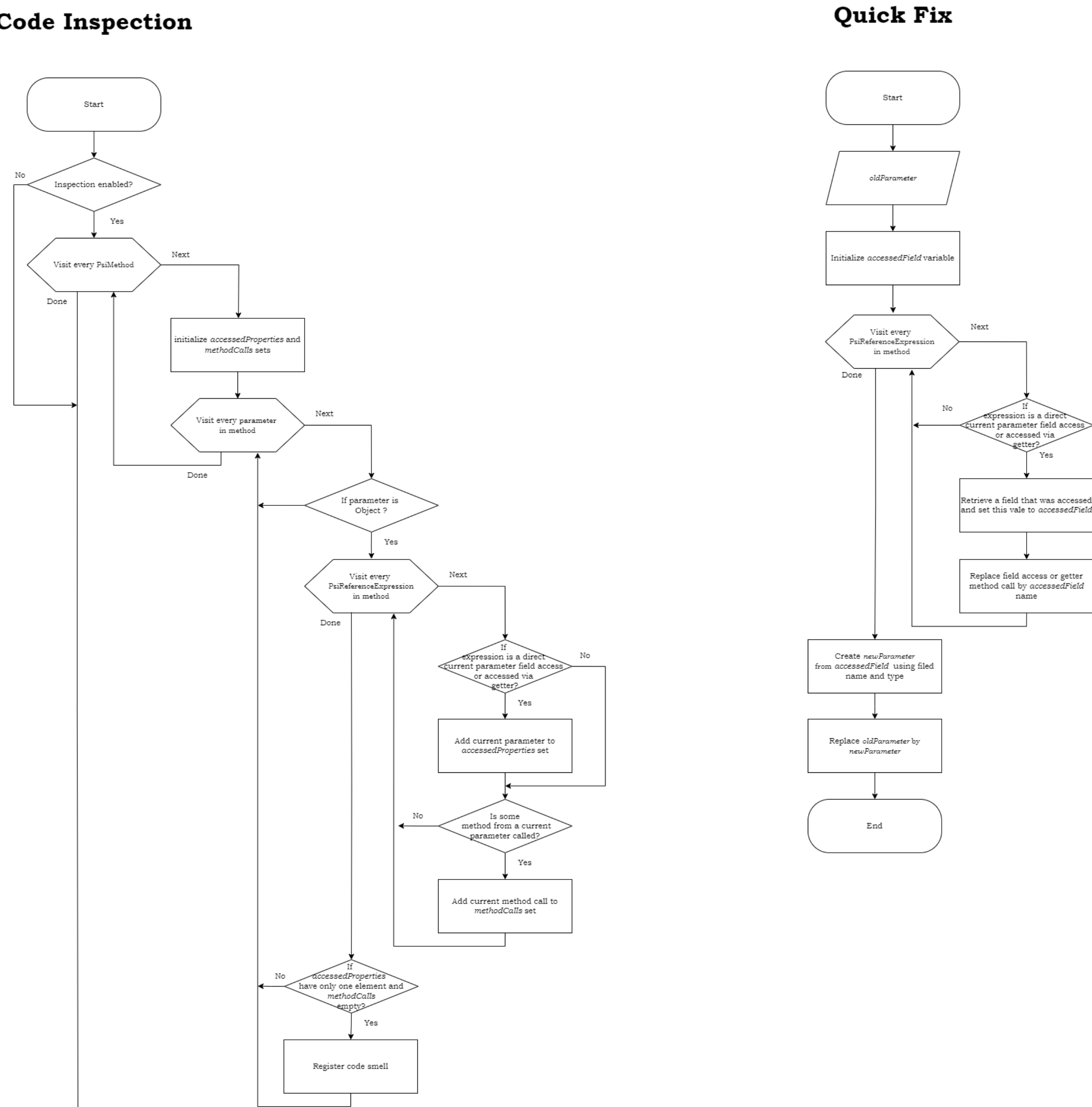


Figure 4.22 – Method parameters check code inspection and quick fix algorithms

4.6.2.2.4 Cyclomatic Complexity Calculation and Refactoring

4.6.2.2.4.1 Algorithm

High cyclomatic complexity should be detected because it indicates excessive branching, resulting in code that is difficult to understand, test and maintain. Complex code increases the likelihood of bugs due to multiple execution paths and makes it difficult for developers to understand and modify

the code correctly. It also makes testing more difficult as it becomes difficult to cover all paths. In addition, high complexity limits the potential for refactoring and modularisation, making the code less adaptable. Early detection of this code smell enables developers to refactor methods into simpler, more maintainable structures, improving code clarity and robustness.

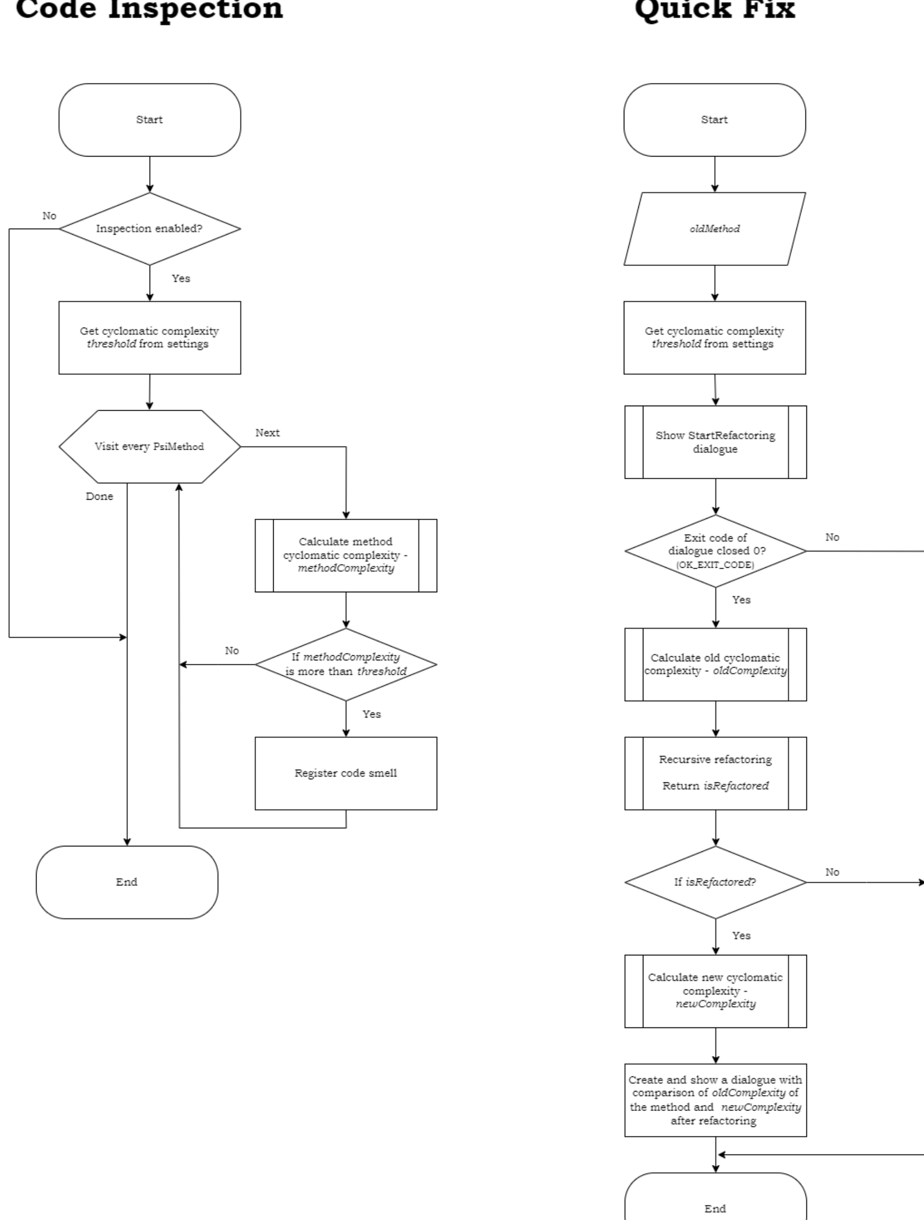


Figure 4.23 – Cyclometic complexity check code inspection and quick fix algorithms

The code inspection algorithm (left side of Figure 4.23) is initiated by checking whether inspection is enabled. If so, it retrieves the pre-defined cyclomatic complexity threshold from the settings to use as a benchmark. The algorithm then systematically visits each method in the code base and calculates its cyclomatic complexity. Methods that exceed the set threshold are flagged, indicating a potential maintenance issue due to their complexity. This process involves subprogram to calculate cyclomatic complexity, shown on Figure 4.24. Cyclomatic complexity quantifies the branching logic within a method, which serves as an indicator of its overall complexity. The

`CyclomaticComplexityVisitor` [30] is a specialised open-source class that calculates this metric by analysing control flow constructs such as conditional statements (`if`, `else`), loops (`for`, `while`) and case statements, all of which contribute to branching in the program flow. This visitor class is crucial for assessing the complexity of methods and identifying areas that may benefit from refactoring.

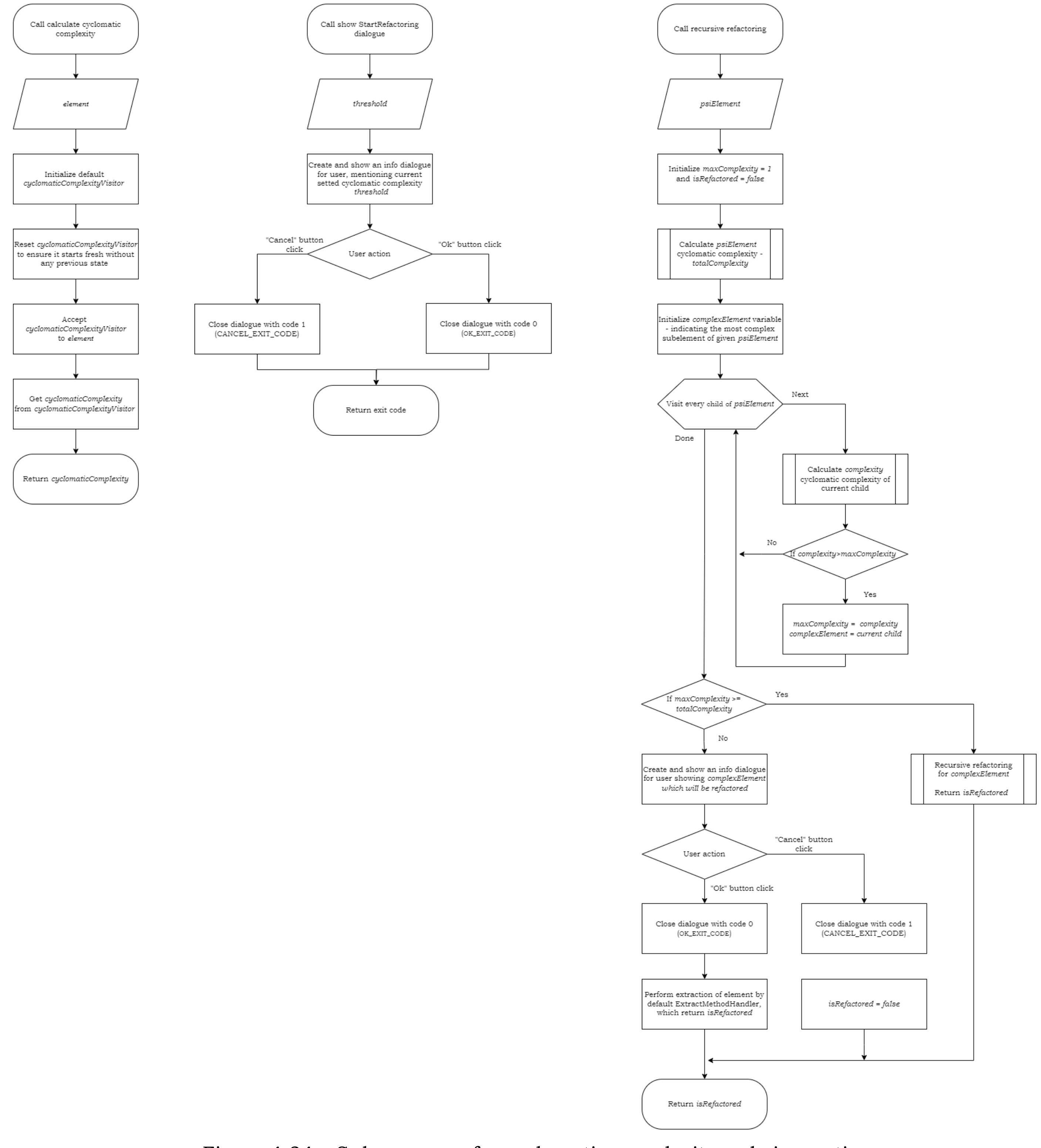


Figure 4.24 – Subprograms for cyclomatic complexity code inspection

The Quick Fix algorithm (right side of Figure 4.23) begins by retrieving the cyclomatic complexity threshold and prompts the user with a `StartRefactoring` dialogue to confirm whether they wish to refactor a complex method. Depending on the user's decision, either the dialogue is closed without further action, or the algorithm proceeds to refactor the method. It first calculates the old complexity of the method and then performs a recursive refactoring to reduce it. After refactoring, it reevaluates the complexity of the method to measure the effectiveness of the changes. The algorithm uses the following subprograms, which are shown on Figure 4.24:

1. *Show Start Refactoring Dialogue*. This subroutine presents the user with a dialogue asking if they wish to proceed with refactoring. This step involves user interaction to confirm the intention to address the complexity problem.
2. *Recursive refactoring*. This subroutine performs recursive refactoring on a method, attempting to reduce its complexity. It breaks down complex methods into smaller ones by identifying the most complex element in the current method.
3. *Create and Show Info Dialogue*. This subroutine displays a dialogue that compares the old and new cyclomatic complexity and informs the user about the effectiveness of the refactoring. It provides feedback on whether the changes have successfully reduced complexity.

In addition, the Quick Fix algorithm involves the use of the `ExtractMethodHandler` [31]. This handler is responsible for extracting parts of a method into a separate method, a common refactoring technique used to reduce complexity. The handler identifies code segments that can be logically grouped and moved into a new method, thereby simplifying the original method and improving code modularity. This approach is particularly effective for managing methods with high cyclomatic complexity, as it breaks down complex logic into more manageable chunks.

4.6.2.2.4.2 Limitations

The current algorithm for addressing cyclomatic complexity identifies methods with high complexity and attempts to refactor them using a quick fix approach. However, the quick fix currently operates by extracting the most complex element from the method. This approach, while reducing the complexity of the original method, does not guarantee that the extracted element will have an acceptable level of cyclomatic complexity after refactoring. Consequently, it may require multiple rounds of applying the quick fix to achieve the desired outcome, which is inefficient and potentially time-consuming.

To improve this process, it would be preferable to perform a more comprehensive refactoring in a single round. This could involve evaluating the complexity of potential extractions in advance and then applying a fix that reduces the complexity of both the original method and the extracted element to acceptable levels. By addressing the issue holistically, the refactoring process would be more efficient, producing cleaner code with fewer iterations and thereby enhancing code maintainability and readability more effectively.

4.6.2.2.5 Long Methods Identification and Refactoring

Long method code is detrimental to the quality of software because it often encapsulates too many responsibilities within a single method, making the method complex and difficult to understand, maintain, and modify. Such methods exhibit high complexity, which indicates a high number of potential

execution paths and, consequently, a higher likelihood of bugs due to the increased difficulty in comprehensively testing the method.

Additionally, long methods hinder code readability and reusability. Long methods present a challenge to developers, particularly those new to the codebase, in that they make it difficult to grasp the functionality quickly. Breaking down long methods into smaller, more focused methods improves modularity and promotes code reuse, aligning with the principles of clean code and effective software engineering practices.

To effectively identify long methods, specific metrics and their thresholds are used:

1. *Nesting depth* (nestingDepth > 4). A depth greater than 4 often indicates overly complex conditional or looping structures within a method, which can make the logic difficult to follow and error-prone.
2. *Number of parameters* (numOfParameters > 6). A method taking more than six parameters can be considered overly complex, suggesting that it might be doing too much by manipulating too many pieces of data.
3. *Lines of code* (linesOfCode > 100). Methods longer than 100 lines are likely to be doing more than one thing, which violates the Single Responsibility Principle. Such methods are prime candidates for refactoring into smaller, more manageable functions.

These thresholds are not arbitrary but are based on empirical research and expert consensus within the field of software engineering, such as guidelines suggested in Robert Martin's “Clean Code” [1].

The code inspection process, shown on left side of Figure 4.25 begins by checking if inspection is enabled. If so, the algorithm iterates through each method in the source code. For each method, it calculates the nesting depth, counts the number of parameters, and measures the lines of code, excluding comments and empty lines. These metrics are then evaluated against the predetermined thresholds, a code smell is registered.

When a code smell is detected, the Quick Fix process can be initiated. Process shown on right side of Figure 4.25. This begins with a dialog box that lets the developer now about current problem and next steps (as shown in the "Show Start Refactoring Dialogue" subprogram on Figure 4.24). If the developer agrees to do changes, the method is then refined in a process called recursive refactoring. This process aims to make the method simpler by shortening its length, reducing the number of its parameters, and decreasing its depth of nesting.

After the refactoring is done, the method is checked again to see if the changes were effective by comparing metrics before and after the refactoring. A comparison dialog box is shown to display the measurements, providing a clear view of how much the method has improved. The refactoring is considered successful if the new measurements are below the original once, indicating that the method's complexity has been reduced.

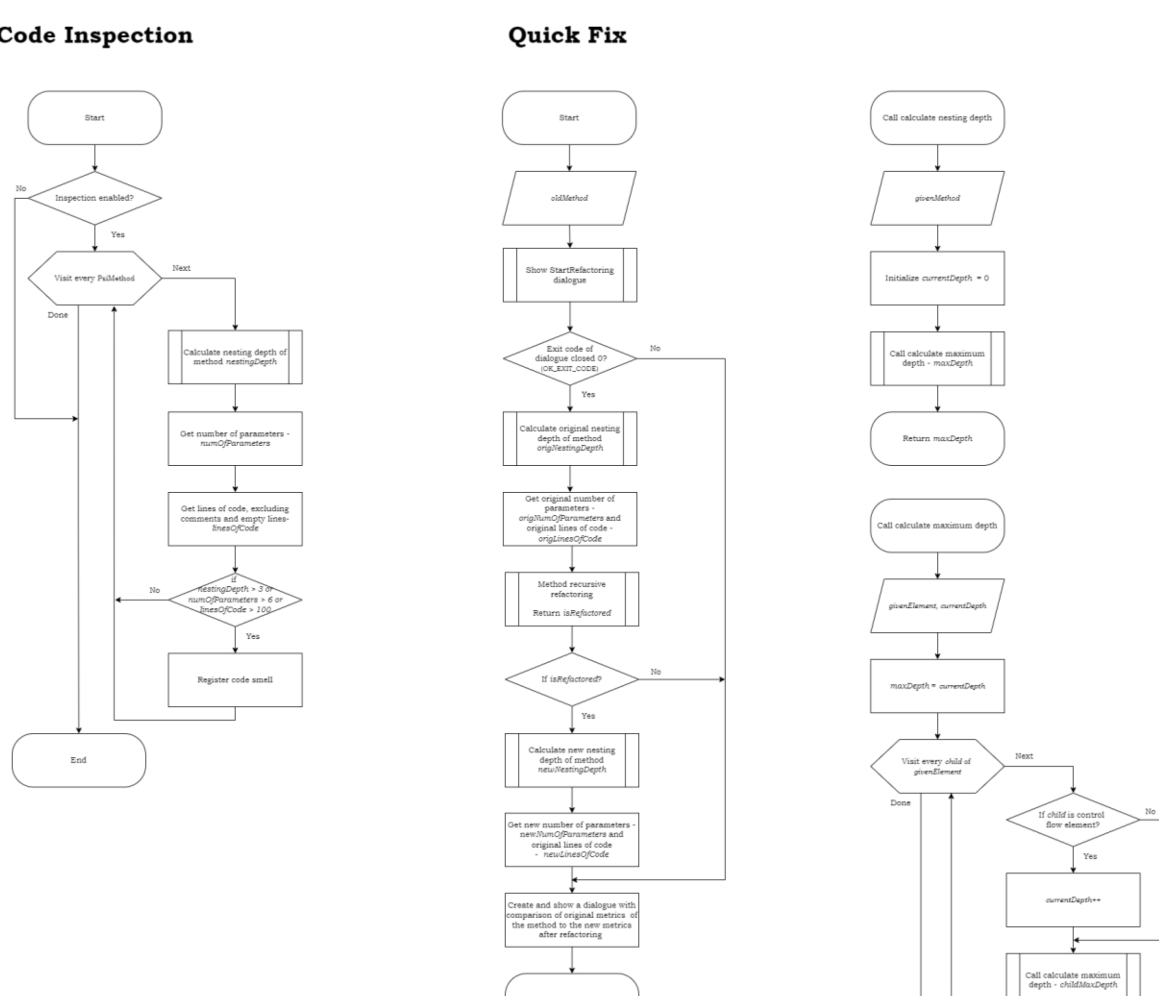


Figure 4.25 – Long method check code inspection and quick fix algorithms

The recursive refactoring subprogram, as depicted on the Figure 4.26, is a method designed to simplify complex software functions by focusing on exceeding specific metric. The initial step in this process is to assess the current complexity of a method. If the nesting depth or lines of code exceeding thresholds, the method is considered for complexity reducing refactoring. This refactoring process detailed on Figure 4.24 focuses on extracting the most complex control flow sub-elements within a method to a private method. The primary goal of this algorithm is to simplify the method by reducing various types of complexity, such as cyclomatic complexity, the length of the method, and nesting depth levels. This approach is particularly valuable because it can be applied across different scenarios where method complexity hinders maintainability and readability. By isolating and refactoring the most intricate parts of the control flow, the algorithm effectively makes the overall code structure simpler and more manageable.

Othervice, if number of parameters exceeds allowed amount, a dialogue is presented to the developer, who is prompted to define a name for a new class, which is referred here as “parametersClassName”. This class is designed to encapsulate all the method's parameters and is created in the same directory as the method. The class includes fields for each parameter, along with constructors and getters. The constructors facilitate the initialisation of all parameters, while the getters provide access to these fields after encapsulation.

Subsequently, the method's original parameters are replaced with a single parameter of the new class type, and the method body is adjusted by replacing the old parameter usages with calls to the new class's getters. This alteration reduces the complexity of the method's interface by reducing the number of parameters.

Furthermore, all instances of the method call throughout the codebase are updated to utilise the new parameter class in place of individual parameters. This ensures that the method's new signature is correctly implemented in all instances where it is called. Upon completion of the changes, the process will mark the method as successfully refactored.

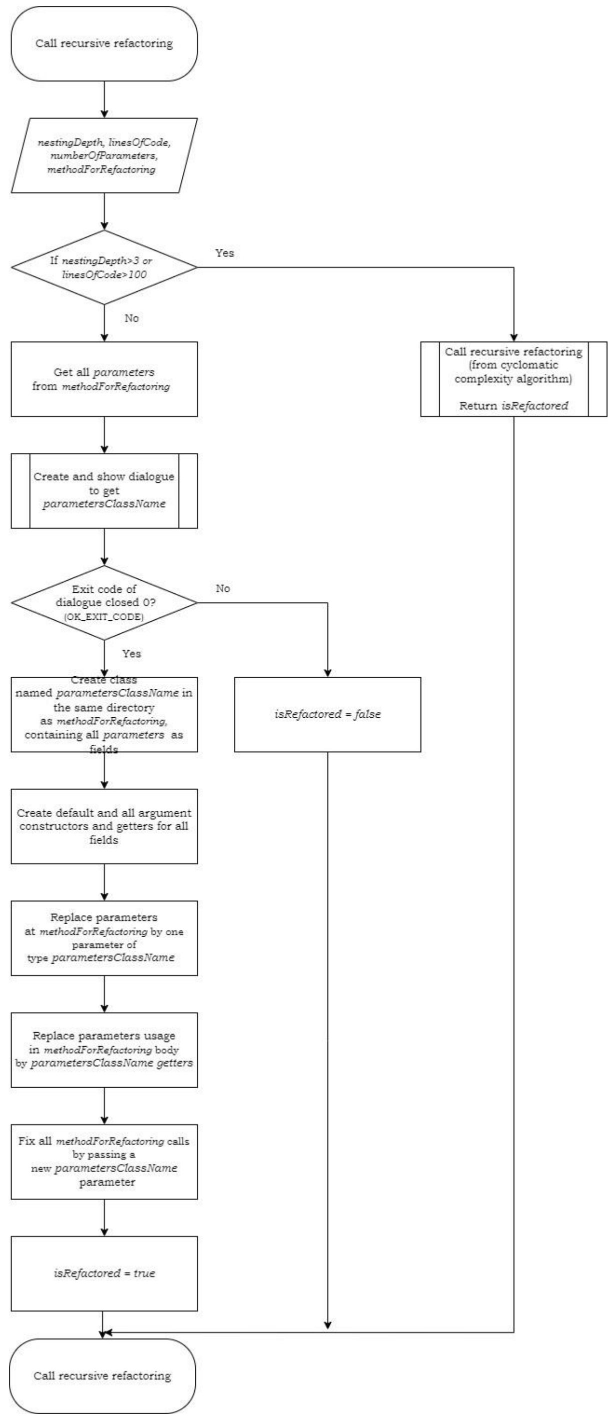


Figure 4.26 – Long method recursive refactoring algorithm

4.6.2.2.6 Constant's Extraction

4.6.2.2.6.1 Algorithm

A notable issue that can degrade both the quality and performance of code is a code smell known as "repeated object creation." This specific problem occurs when the same object, with identical parameters, is created multiple times throughout the code instead of using a single instance or a constant. This redundancy not only consumes extra memory but also increases the processing overhead, as each creation involves constructor calls and memory allocations.

The repercussions of repeated object creation are significant:

1. *Performance Impact.* Each unnecessary instantiation increases the runtime overhead because of the memory allocation and the constructor execution involved. In critical sections of the code or within loops, this can lead to noticeable slowdowns.
2. *Increased Memory Usage.* Creating multiple objects when a single object or a constant could be used increases the program's memory footprint. This is particularly problematic in environments with limited resources, where optimal memory usage is crucial.
3. *Reduced Maintainability.* When the same object is created repeatedly, any modification to the creation logic must be replicated across all occurrences. This not only increases the chance of errors during updates but also complicates the code, making it harder to read and maintain.

The detection algorithm, shown on left side of Figure 4.27, initiates by verifying whether the inspection is enabled. Then inspect the code and captures each new object instantiation, from which it generates a unique key based on the constructor and its parameters. This key serves to identify and group identical object creations. These keys and their associated instances are stored in a map, which facilitates the aggregation and rapid retrieval of similar instances. Following the completion of the analysis, the algorithm evaluates the collected data in order to identify keys that are linked to multiple object creation instances. These instances indicate potential inefficiencies within the code. For each key with multiple associated instances, a code smell is registered. This enables developers to identify the exact locations in the code that may require refactoring.

In the event of the detection of duplications in the creation of objects, the tool offers the option of implementing automated corrections with the objective of streamlining the code. The correction process, shown on reight side of Figure 4.27, commences with the verification that the repeated object creations do not already correspond to an existing constant. If no existing constant matches the identified objects, the tool proceeds to create one.

The process of creating a new constant involves the generation of a standardised name that reflects the object's type and usage. Subsequently, the newly created constant is strategically inserted into the appropriate class, at the beginning of constants declaration section. This placement is of great importance, as it affects the constant's scope and accessibility throughout the application.

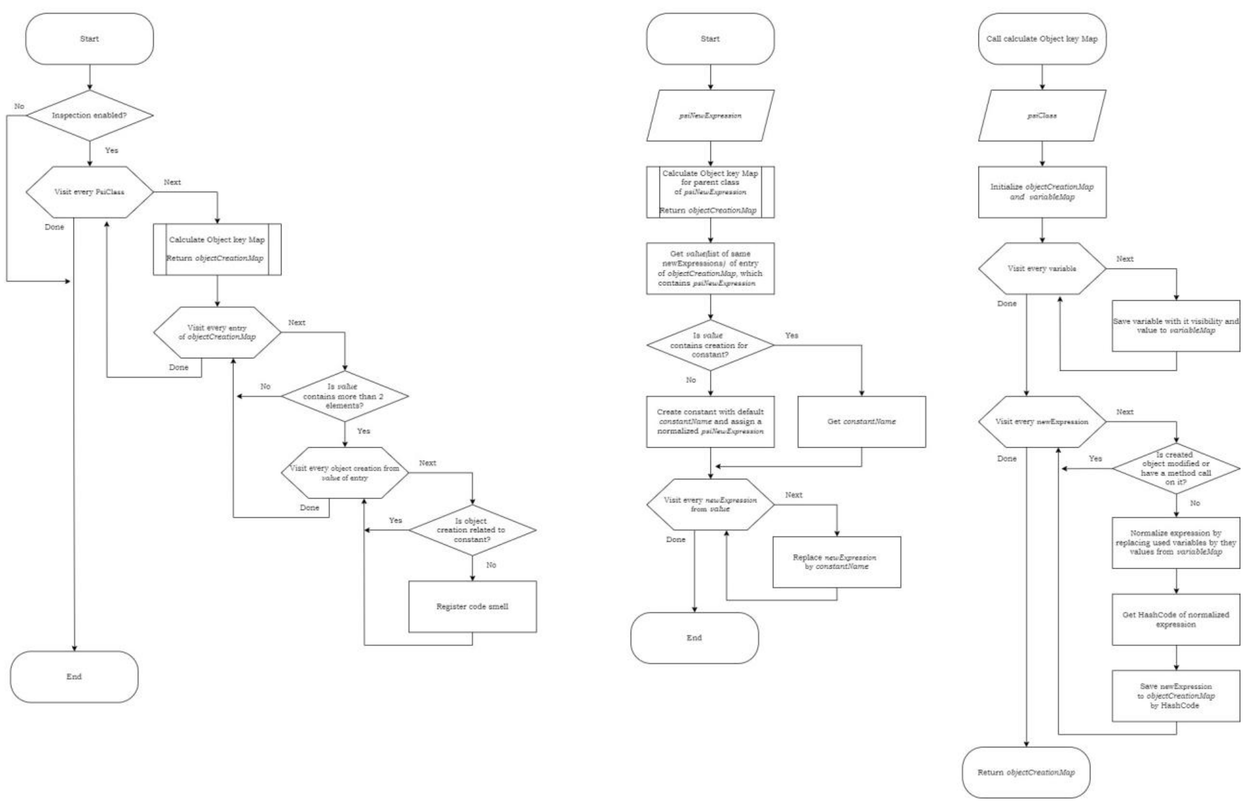


Figure 4.27 – Repeated object creation code inspection and quick fix algorithms

Once the new constant has been established, the tool proceeds to replace all instances of the repeated object creation in the code with references to this new constant. This step is of critical importance, as it not only reduces the memory usage of the application by eliminating redundant object creations, but also simplifies the code, making it cleaner and easier to maintain. By referencing a single constant, future modifications to the object instantiation are concentrated in a single location, which significantly reduces the likelihood of errors during updates and ensures consistency across the application.

4.6.2.2.6.2 Limitations

The algorithm has certain limitations that could affect its applicability and effectiveness in various coding scenarios. It is important to consider two key limitations of the algorithm:

1. *The scope of object creation analysis.* The current implementation of the algorithm is limited to the analysis of object creations made directly through the new keyword. This limitation implies that object creations facilitated by alternative methods, such as factory methods, builders, or dependency injection frameworks, are not analysed or corrected. Given that these patterns are prevalent in modern Java applications, particularly those that adhere to sound design principles such as loose coupling and high cohesion, the algorithm may fail to identify significant opportunities for optimisation in these areas.
2. *The impact of refactoring on semantic meaning.* The replacement of multiple object creations with a single constant may occasionally result in alterations to the semantic meaning or intended logic of the original code.

For example, the original developer may have deliberately created multiple instances of an object for reasons such as ensuring thread safety or maintaining a consistent state with each use. The algorithm does not account for such nuances, which may result in unintended side effects following the refactoring.

These limitations serve to emphasise the importance of an understanding of the specific contexts and patterns utilised in the application code prior to the application of automated refactoring tools. These considerations underscore the necessity for manual review and potential post-refactoring adjustments to ensure that the changes align with the original design intentions and do not introduce new issues.

4.6.2.3 Test Code Smells

4.6.2.3.1 Test Naming Convention Enforcement

4.6.2.3.1.1 Algorithm

Properly naming test methods is crucial for several reasons. Firstly, it improves readability and maintainability. Test names that are clear and descriptive help developers understand what each test is supposed to validate, making the tests easier to maintain and modify in the future. Secondly, consistency is important. Consistency in naming conventions across test methods helps create a uniform codebase, which improves readability and makes it easier for teams to collaborate. In contrast, inconsistent naming can lead to confusion and reduce the efficiency of development and debugging processes. Furthermore, adhering to testing standards is essential for maintaining code quality and professionalism. Incorrect naming could hinder the effectiveness of testing frameworks or tools, leading to missed tests during automated test runs.

The Code Inspection algorithm on the left of Figure 4.28 is designed to detect test methods that do not adhere to a specified naming pattern. It retrieves the expected naming pattern from the settings and then iterates over every method in the codebase, checking if it has a JUnit4, JUnit5 or TestNG annotation. It then iterates over every method in the codebase, checking for JUnit4, JUnit5 or TestNG annotations for each method. If the method is annotated, the algorithm checks if the method's name matches the specified naming pattern. If the method's name does not match the naming pattern, the algorithm will register a code smell, indicating that the method should be renamed for consistency.

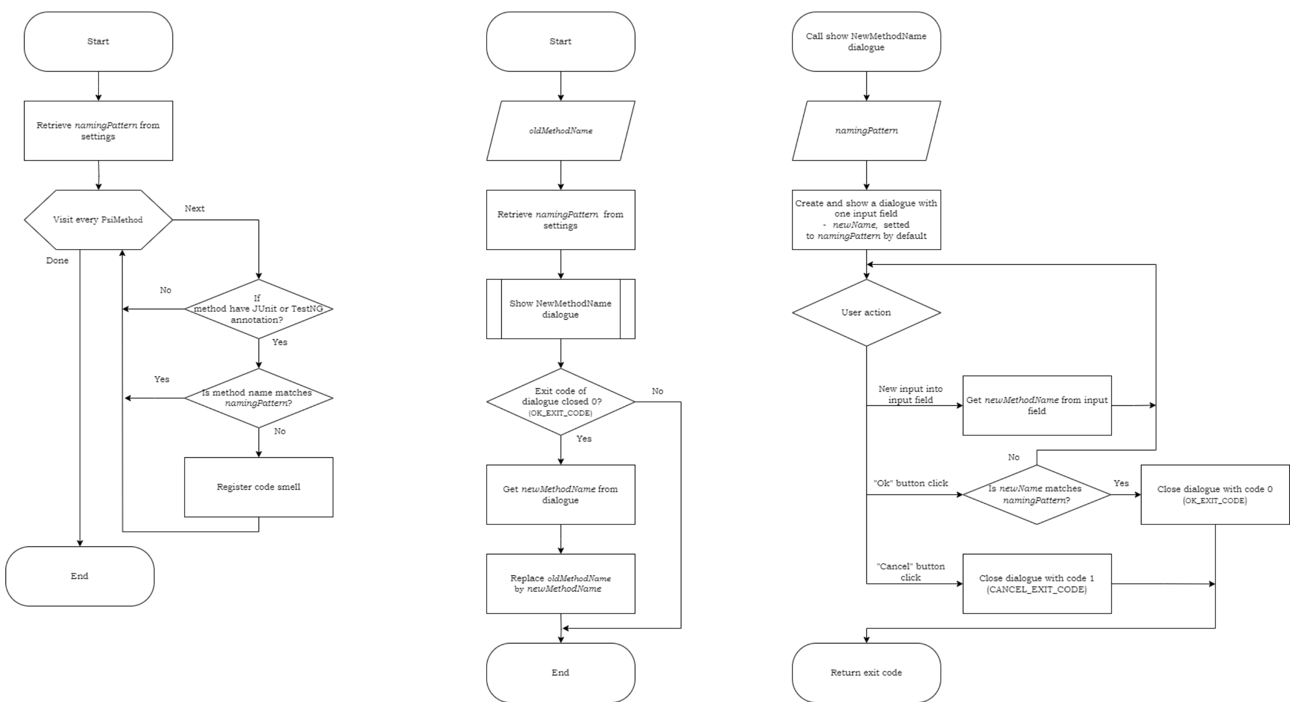


Figure 4.28 – Test naming convention code inspection and quick fix algorithms

The Quick Fix algorithm on the right of Figure 4.x3 is designed to rename test methods that do not conform to a specified naming pattern. The algorithm starts by identifying the method to be renamed and retrieving the naming pattern from the settings. It then displays a NewMethodName dialogue, asking the user to enter a new name for the method. The default is the specified naming pattern. The user is presented with an input field that has the suggested new name, based on the naming pattern, already entered. If the user provides a new name that matches the pattern and clicks "OK", the method is renamed to the new method name. If the new name does not follow the pattern, the button "OK" will be disabled. If the user clicks "Cancel", the dialogue will close without changing the method's name.

4.6.2.3.1.2 Limitations

The current algorithm identifies a method as a test method if it is annotated with JUnit4, JUnit5, or TestNG annotations. While this approach is effective for many test methods, it has notable limitations:

1. *Annotation support.* The algorithm's reliance on specific annotations may inadvertently exclude other commonly used testing frameworks or custom test annotations utilized within organizations. This exclusion could result in missed test methods that should adhere to the specified naming convention.
2. *Custom annotations*. Many codebases include custom annotations for testing purposes or employ third-party testing frameworks that are not JUnit or TestNG. These will not be recognised by the current algorithm, which could result in inconsistencies in test method naming conventions.
3. *Framework evolution.* As new testing frameworks or versions emerge, these will almost certainly employ different annotations. It is therefore essential to regularly update the algorithm to maintain accuracy. This creates a significant maintenance overhead and increases the risk of failing to identify test methods correctly.

The following solutions for explained limitations are potential options:

1. *Configurable annotations*. Allowing users to configure the list of annotations considered as test annotations will increase the algorithm's flexibility. This will enable teams to adapt it to their specific needs, including support for custom or third-party annotations.
2. *Annotation interface*. The algorithm should check if a method is annotated with any annotation that extends a common test annotation interface. This approach will cover a broader range of testing annotations and accommodate future extensions or frameworks adhering to a shared interface.
3. *Annotation patterns*. Using regular expressions or patterns can be used to identify potential test annotations, allowing the algorithm to capture a wider variety of annotations without explicitly listing them. This will ensure greater flexibility and adaptability to changes in naming conventions or new frameworks.

Implementing these solutions will make the algorithm more robust and versatile, ensuring that all relevant test methods are identified and validated against the desired naming conventions.

5 Project Management

5.1 Methodology

The development of a refactoring plugin requires a methodology that supports rapid iterations, responsiveness to user feedback, and the ability to manage a complex feature set. Given these requirements, an agile, iterative approach is tailored for this project. This section describes the adaptation of the methodology, focusing on concrete steps and practices appropriate to the single-developer scenario.

Reasons for choosing an agile methodology:

1. *Flexibility.* Agile is chosen for its ability to accommodate the evolving nature of plugin development, where user feedback may require changes to the feature set or implementation approach.
2. *Incremental development.* Development of the plugin will be incremental, allowing early delivery of a base version and subsequent enhancements based on priority and user input.
3. *User-centric design and feedback.* Agile's emphasis on user engagement is critical to developing a plugin that meets the real-world needs of developers, ensuring that features are intuitive and add value to their coding experience. User engagement in this context will imply actively involving software developers who would potentially use the plugin in providing feedback, iterating on design improvements, and ensuring that the features align with their coding workflows and preferences.

As Agile was chosen, the process (shown on Figure 5.1) will be organized according to the following rules:

1. *Sprint plannings.* Each sprint will prioritise a set of features or enhancements to be developed, based on the plugin roadmap and feedback. Feature implementation is broken down into manageable tasks, each of which is estimated and prioritised.
2. *Solo stand-ups for self-review.* Daily self-reviews will replace traditional stand-ups, and will be used to assess progress, identify blockers, and plan the day's tasks to keep development on track.
3. *Sprint reviews for feature evaluation.* At the end of each sprint, a review session will assess the implemented features against the goals, evaluating functionality, usability and integration within the IDE environment.
4. *Iterative testing and refinement.* Continuous testing will be integrated throughout the sprints, focusing on unit testing for individual components and integration testing for each function. Feedback from beta testers will be used for iterative refinement on last sprints.

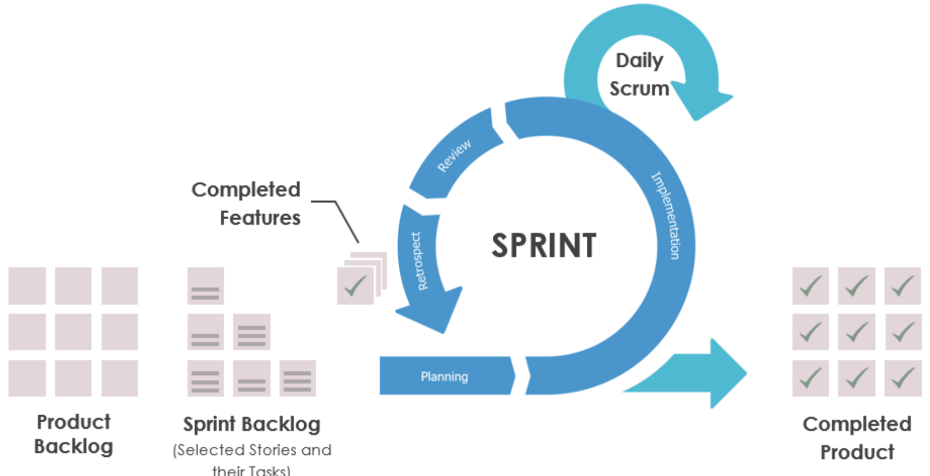


Figure 5.1 – Agile development life cycle

The following tools and techniques will be used to implement the described lifecycle:

1. *Agile board.* The use of a digital agile board organises tasks related to different plugin features, bug fixes and documentation, improving visibility and task management.
2. *Estimation with story points.* Tasks are estimated using story points, facilitating prioritisation based on complexity and effort. This helps with sprint planning and ensures a balanced workload.
3. *Feedback loops and beta testing.* Working with a group of beta testers provides direct feedback on the plugin's usability, functionality and performance. This feedback loop is essential for prioritising refinements and new features.

5.2 Project Planning

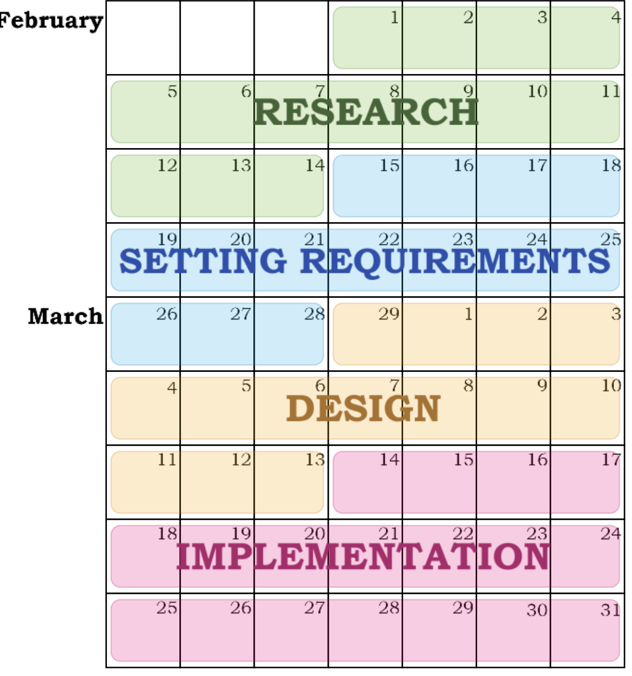
5.2.1 Scheduling

The development of any product requires detailed planning and execution. Given the scope of the project from 1 February to 31 May, the following plan, shown on Figure 5.2, outlines the key phases, including analysis, requirements gathering, implementation, testing and evaluation, with an emphasis on agile practices tailored to a single developer scenario:

1. *February.* Analysis and Requirements Gathering.
    a. Market and theoretical research (Feb 1 - 14):
        i. Conduct market research to analyze existing refactoring plugins, focusing on their features, limitations, and user feedback.
        ii. Perform a theoretical review of software engineering literature to identify best practices for effective refactoring.
    b. *Sprint 0.* Requirements Gathering and Survey (Feb 15 - 28):
        i. Define a set of requirements for the plugin.
        ii. Conduct a survey within the group of developers to prioritize the features based on user demand.
2. *March-April.* Design and Initial Development.
    a. *Sprint 1.* System design (Feb 29 - Mar 13):
        i. Develop a system architecture for the plugin, focusing on scalability, performance, and ease of use within the IntelliJ IDEA ecosystem.
    b. *Sprint 2 - 4.* Core Development (Mar 14 – Apr 24):

i. Design and implementing features of the plugin, starting with the most critical refactoring operations as identified from the survey.

3. *May.* Testing, User Feedback, and Refinement.
   a. *Sprint 5.* Testing and Feedback (Apr 25 – May 8).
      i. Perform additional manual testing to ensure the stability and reliability of the plugin.
      ii. Release a beta version for a group of users, and gather feedback on usability and functionality.
   b. *Sprint 6.* Evaluation and Final Refinements (May 9 – May 22).
      i. Analyze feedback from beta testing to identify areas for improvement.
      ii. Make necessary adjustments to enhance functionality, fix bugs, and improve the overall user experience based on real user evaluations.
   c. *Sprint 7.* Documentation and Release Preparation (May 22- 31).
      i. Develop documentation for the plugin, including installation instructions, user guides, and examples of refactoring operations. This documentation aims to assist users in maximizing the plugin's benefits and ensuring a smooth integration into their development workflow.
      ii. Prepare the plugin for official release, including finalizing the deployment package and publishing the plugin on the JetBrains Marketplace.
      iii. Conduct a final review and testing cycle to ensure the plugin and its documentation are free of errors and ready for public use.



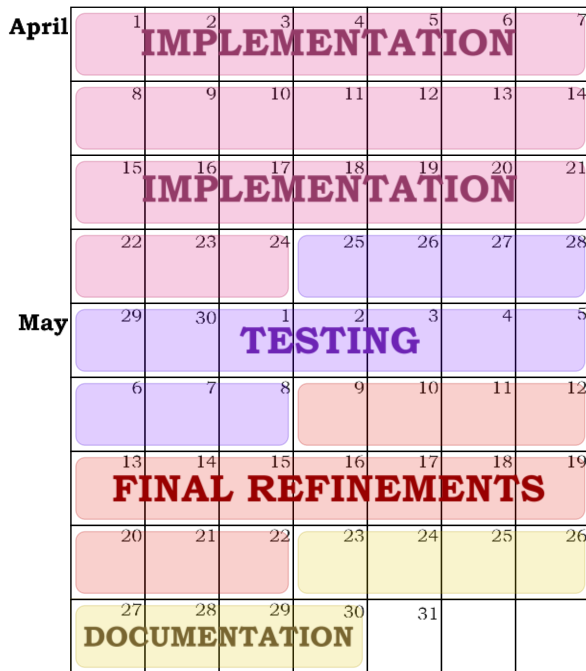


Figure 5.2 – Project planning calendar

5.2.2 Task Prioritization

Time constraints made it clear that not all the features could be implemented within the timeframe outlined in the project plan. In order to

prioritise effectively, a survey was conducted to gauge the needs of potential users, asking them to rate the importance of each proposed feature on a scale from 0 (not needed at all) to 5 (very important). Results of this survey, shown in Figure 5.3, facilitated an informed prioritisation process.

Cyclic Dependency detection	God Components detection	Scattered Functionality detection (repeating code blocks in different parts of module)	Detect reference object comparison instead of conte[illegible] comparison (== instead of Objects.equals())	Detect content enums comparison instead of [illegible] ==)	Detect and refactor long statements (more than 4 conditions)	Detect and suggest refactoring options for long methods	Detect and extract to constants magic numbers and strings	Extract constants from code (repeated objects usage)	Calculate and suggest refactoring options for methods that exceed allowed cyclomatic complexity	Enforce naming convention for test methods (should_when"test MethodName"_given)	Identify empty or ignored tests	Ensure parameters for tested methods have prefixes "given" or "actual"	Ensure asserted result variables have prefix "expected"	Boolean Expression Optimization	Boolean Expression Optimization	Object method parameters check	Method naming conventions
5	4	5	3	5	4	5	5	5	4	5	5	4	4	5	5	5	3
5	3	2	2	5	3	3	3	4	3	3	2	2	2	1	1	5	3
3	4	5	4	3	2	4	4	3	4	5	1	4	1	3	3	5	3
1	4	5	5	3	3	3	2	3	1	2	2	4	1	5	5	4	3
5	5	5	4	4	3	3	5	3	4	5	4	2	4	3	3	4	3
1	3	4	3	4	3	4	3	3	2	5	4	4	3	5	5	4	5
4	3	4	4	3	3	4	4	5	5	4	3	1	1	4	4	4	5
1	3	5	3	5	5	5	5	5	5	5	5	4	5	4	4	4	4
4	3	5	2	5	5	5	5	5	4	5	4	2	2	2	2	5	3
3	3	5	5	5	4	5	5	5	4	5	4	4	4	3	3	4	4
3	5	5	1	1	4	5	3	3	5	5	2	4	4	5	5	3	4
3	3	4	5	4	4	4	4	4	4	1	5	4	4	3	3	2	5
5	5	5	5	2	4	4	4	4	4	2	1	1	1	4	4	5	3
5	4	5	4	4	4	3	3	4	4	4	5	2	2	4	4	3	4
5	5	3	5	4	4	5	5	4	5	5	5	4	5	5	5	4	4
5	5	5	5	5	2	2	2	2	5	4	1	1	1	2	2	5	3
3,625	3,875	4,5	3,75	3,875	3,5625	4	3,875	3,875	3,9375	4,0625	3,3125	2,9375	2,75	3,625	3,625	4,125	3,6875

Figure 5.3 – User survey results

Notion [32] was chosen to create the Agile board due to its free and customizable nature, offering the flexibility to tailor Agile boards to fit specific project needs without incurring additional costs. The average importance of each feature was calculated based on average value from the survey and displayed on the project's agile board, with tasks categorized as software design, interface, customization and implementation, architecture or testing code smells, based on their nature.

In prioritizing tasks for the project's agile board, the average importance of each feature was calculated, providing valuable insights into user preferences. However, certain categories have also emerged as having higher priority than others. Software design, interface, customization, evaluation, documentation and release actions tasks have also been assigned the highest priority level of 5. This decision stems from the understanding that these aspects are foundational and essential for the overall success of the project. Regardless of specific user demands, these elements must be implemented to ensure the functionality and usability of the refactoring plugin. Conversely, tasks related to addressing code smells have been prioritized based on user survey feedback. While it's crucial to incorporate as many code smell tasks as possible, it's also recognized that not all can be addressed within the project scope. Therefore, prioritization within this category is guided by the need to balance user demands with project constraints, aiming to maximize the overall effectiveness of the plugin by delivering the most impactful improvements.

All tasks have been sorted by priority and displayed on Figure 5.4 in the project documentation. In total, there are 31 tasks, which have been estimated at 148 story points based on the size of each task.

agile board ···

Name	Estimation	Priority	Type	Status
RPD-1 Design System Archite OPEN	8	5	Software Design	Backlog
RPD-2 Project Setup	3	5	Software Design	Backlog
RPIF-1 Design and Develop Project Analys	5	5	Interface	Backlog
RPC-1 Implement customization options	5	5	Customization	Backlog
RPE-1 Manual Testing	5	5	Evaluation	Backlog
RPREL-1 Release Beta Version	3	5	Release Actions	Backlog
RPE-2 Collect and Analyze Beta Testing Fe	5	5	Evaluation	Backlog
RPAJ-1 Implement Necessary Adjustments	8	5	Implementation Smell	Backlog
RPD-1 Develop Plugin Documentation	5	5	Documentation	Backlog
RPREL-2 Prepare Plugin for Official Relea	3	5	Release Actions	Backlog
RPREL-3 Perform Official Release	2	5	Release Actions	Backlog
PRE-3 Conduct Final Review and Testing	3	5	Evaluation	Backlog
RPM-1. OOP Metrics Calculation	8	4.8	Metrics	Backlog
RPA-3. Identify Scattered Functionality	8	4.5	Architecture Smell	Backlog
RPI-5. Object method parameters check	5	4.12	Implementation Smell	Backlog
RPT-1. Enforce naming convention for tes	3	4.1	Testing Smell	Backlog
RPI-6. Identify and refactor long methods	8	4	Implementation Smell	Backlog
RPA-2. Detect Cyclic Dependency betwee	5	3.94	Architecture Smell	Backlog
RPI-9. Extract constants from code	5	3.88	Implementation Smell	Backlog
RPI-8. Detect and extract to constants mag	5	3.88	Implementation Smell	Backlog
RPA-1. Identify God Components	5	3.88	Architecture Smell	Backlog
RPI-2. Detect content enums comparison i	2	3.88	Implementation Smell	Backlog
RPI-1. Detect reference object comparison	2	3.75	Implementation Smell	Backlog
RPI-7. Method naming conventions	8	3.69	Implementation Smell	Backlog
RPI-10. Calculate and refactor cyclomatic	8	3.63	Implementation Smell	Backlog
RPI-4. Boolean Expression Optimization	5	3.63	Implementation Smell	Backlog
RPI-3. Detect and refactor complex condit	5	3.56	Implementation Smell	Backlog
RPT-2. Identify and handle empty and ign	3	3.31	Testing Smell	Backlog
RPT-3. Ensure parameters for tested meth	5	2.93	Testing Smell	Backlog
RPT-4. Ensure asserted result variables ha	3	2.75	Testing Smell	Backlog
+ New				
COUNT 31	SUM 148			

Figure 5.4 – Prioritized tasks

The next step involved sprints planning. In total, 7 sprints were planned (excluding Sprint 0, where requirements were established and the backlog was compiled). A capacity of 13-16 story points per sprint is selected, tailored to the single developer scenario. This choice ensures a manageable workload, allowing the developer to focus on completing tasks effectively within the sprint timeframe. It strikes a balance between task complexity and achievable goals, providing flexibility to adapt to evolving requirements while maintaining productivity. The task distribution across sprints is illustrated in Figure 5.5.

agile board

↓ Priority

Sprint 1 3

Name	Estimation	Priority	Type	Status	Sprint
RPD-1 Design System Architecture	8	5	Software Design	Backlog	Sprint 1
RPD-2 Project Setup	3	5	Software Design	Backlog	Sprint 1
RPIF-1 Design and Develop Project Analys	5	5	Interface	Backlog	Sprint 1
+ New					
COUNT 3	SUM 16				

Sprint 2 3

Name	Estimation	Priority	Type	Status	Sprint
RPC-1 Implement customization options	5	5	Customization	Backlog	Sprint 2
RPM-1. OOP Metrics Calculation	8	4.8	Metrics	Backlog	Sprint 2
RPT-1. Enforce naming convention for tes	3	4.1	Testing Smell	Backlog	Sprint 2
+ New					
COUNT 3	SUM 16				

Sprint 3 2

Name	Estimation	Priority	Type	Status	Sprint
RPA-3. Identify Scattered Functionality	8	4.5	Architecture Smell	Backlog	Sprint 3
RPI-6. Identify and refactor long methods	8	4	Implementation Smell	Backlog	Sprint 3
+ New					
COUNT 2	SUM 16				

Sprint 4 3

Name	Estimation	Priority	Type	Status	Sprint
RPI-5. Object method parameters check	5	4.12	Implementation Smell	Backlog	Sprint 4
RPI-10. Calculate and refactor cyclomatic	8	3.94	Implementation Smell	Backlog	Sprint 4
RPI-2. Detect content enums comparison i	2	3.88	Implementation Smell	Backlog	Sprint 4
+ New					
COUNT 3	SUM 15				

Sprint 5 4

Name	Estimation	Priority	Type	Status	Sprint
RPE-1 Manual Testing	5	5	Evaluation	Backlog	Sprint 5
RPREL-1 Release Beta Version	3	5	Release Actions	Backlog	Sprint 5
RPI-9. Extract constants from code	5	3.88	Implementation Smell	Backlog	Sprint 5
RPI-1. Detect reference object comparison	2	3.75	Implementation Smell	Backlog	Sprint 5
+ New					
COUNT 4	SUM 15				

Sprint 6 2

Name	Estimation	Priority	Type	Status	Sprint
RPE-2 Collect and Analyze Beta Testing Fe	5	5	Evaluation	Backlog	Sprint 6
RPAJ-1 Implement Necessary Adjustments	8	5	Implementation Smell	Backlog	Sprint 6
+ New					
COUNT 2	SUM 13				

Sprint 7 4

Name	Estimation	Priority	Type	Status	Sprint
RPD-1 Develop Plugin Documentation	5	5	Documentation	Backlog	Sprint 7
RPREL-2 Prepare Plugin for Official Relea:	3	5	Release Actions	Backlog	Sprint 7
RPREL-3 Perform Official Release	2	5	Release Actions	Backlog	Sprint 7
PRE-3 Conduct Final Review and Testing	3	5	Evaluation	Backlog	Sprint 7
+ New					
COUNT 4	SUM 13				

No Sprint 10

Name	Estimation	Priority	Type	Status	Sprint
RPI-8. Detect and extract to constants mag	5	3.88	Implementation Smell	Backlog	
RPA-1. Identify God Components	5	3.88	Architecture Smell	Backlog	
RPI-7. Method naming conventions	8	3.69	Implementation Smell	Backlog	
RPA-2. Detect Cyclic Dependency betweer	5	3.63	Architecture Smell	Backlog	
RPI-4. Boolean Expression Optimization	5	3.63	Implementation Smell	Backlog	
RPI-3. Detect and refactor complex condit	5	3.56	Implementation Smell	Backlog	
RPT-2. Identify and handle empty and ign	3	3.31	Testing Smell	Backlog	
RPT-3. Ensure parameters for tested meth	5	2.93	Testing Smell	Backlog	
RPT-4. Ensure asserted result variables ha	3	2.75	Testing Smell	Backlog	
+ New					
COUNT 10	SUM 44				

Figure 5.5 – Tasks distributes on sprints

As shown in the figures, it wasn't always possible to include tasks with the highest priority in each sprint. The objective is to complete as many tasks as possible, thus necessitating the addition of lower-priority tasks to the sprints to maintain a capacity of 13-16 story points. This approach ensures optimal utilization of sprint capacity while aiming for a balanced distribution of tasks. It allows us to maximize productivity and make progress toward project objectives effectively.

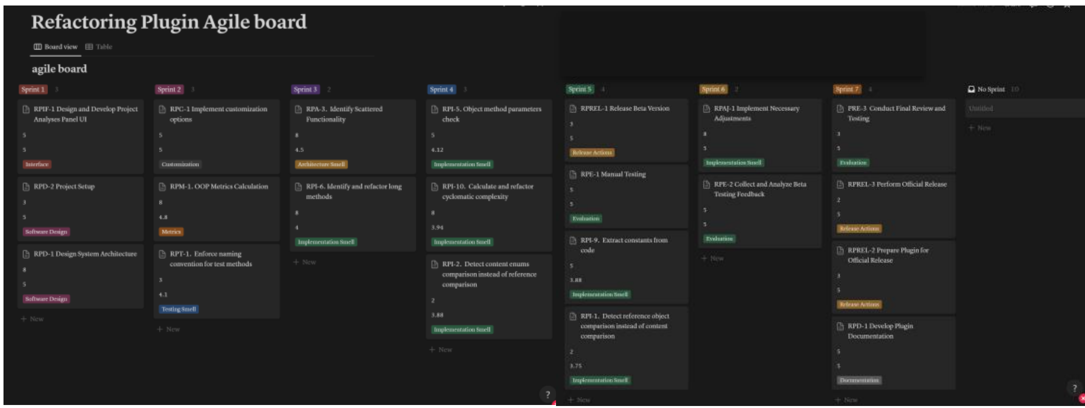


Figure 5.6 – Agile board before the first sprint

Before commencing the first sprint, the project board's status is depicted in Figure 5.6. The backlog comprises 9 remaining tasks, totaling 44 story points. Planning for each sprint includes the allocation of 2-4 tasks. As progress ensues, tasks will transition between the columns of "TODO," "In Progress," and "Done" on the board.

5.3 Monitoring and Evaluation Plan

To ensure successful development, a monitoring and evaluation plan is implemented. This plan is designed to track progress against project goals, ensure adherence to the agile methodology, and facilitate continuous improvement based on feedback and test results.

In a single development scenario, self-monitoring becomes paramount to track progress and ensure adherence to the project timeline. The absence of a team underscores the importance of structured, self-imposed checkpoints and the use of automation to maintain project momentum and quality. Monitoring plan consists of:

1. *Agile board for task management.* An agile digital board, previously shown in Figure 5.4, is used to manage tasks and sprints. This tool will help organise tasks, track progress, and set priorities, ensuring that the developer can maintain focus and efficiency.
2. *Sprint reviews.* In the absence of a team, a self-assessment is carried out on a sprint basis. This involves reviewing completed tasks, assessing progress towards sprint goals, and adjusting upcoming tasks based on current priorities and any unforeseen challenges.

Evaluation strategies must be carefully designed to maximise the resources of the individual developer, using external feedback and automated testing to ensure that the plugin's functionality and user experience meet high standards. Evaluation plan includes the next points:

1. *Automated testing.* Emphasise automated unit and integration testing to ensure the stability and functionality of the plugin. Automated

testing is essential for a single developer to effectively manage code quality without the need for manual testing resources on every iteration. It would be enough one round of manual testing on final stages.

2. *Beta testing with real users.* Release a beta version of the plugin to a small group of users. Collect feedback through surveys or direct communication to gain insight into the plugin's usability, functionality and any issues from an end-user perspective.

3. *Iterative feedback integration.* Analyse user feedback and systematically incorporate necessary changes into the development process. Given the solo nature of the project, prioritisation will be key, focusing on adjustments that will have the most significant impact on usability and performance.

4. *Project retrospective.* At the end of the development cycles, a retrospective analysis should be performed to reflect on the project's successes and areas for improvement. In addition, the documentation must be reviewed to ensure that it is clear, accurate and easy to use, and make any necessary adjustments based on user feedback.

For an individual developer, this monitoring and evaluation plan emphasises efficiency, automation and continuous improvement. By using agile methodologies, automated tools and user feedback, the developer can overcome the challenges of solo development while ensuring the delivery of a quality product.

5.4 Executing the Plan

A total of 104 story points were planned for the 7 sprints. The distribution of these points across the sprints was designed to ensure a balanced workload and consistent progress. By the end of the project, 104 story points had been successfully completed, as planned. To illustrate the process of development, a burndown chart was created for each of the sprints, as shown in Figure 5.7.

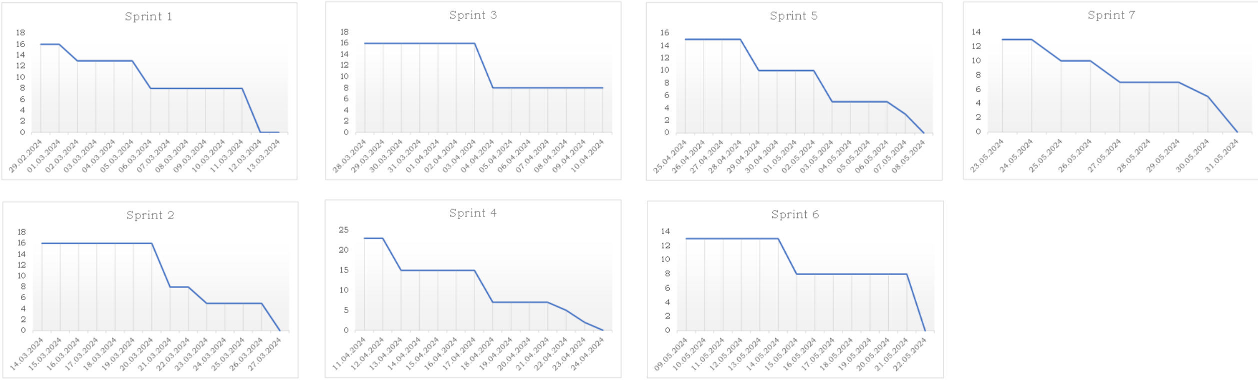


Figure 5.7 – Sprints burndowns

The velocity, which is the measure of how many story points could be completed in a sprint, varied slightly across the sprints, but generally showed a steady trend. The average velocity for the project was approximately 14,86 story points per sprint. The consistency in velocity indicates that it was possible to maintain a stable pace of work, adjusting to challenges and continuously improving the workflow.

A significant challenge encountered during the project execution was a delay in Sprint 3. The primary cause of this delay was the implementation of a

complex task related to the "Long Method code smell inspection." This task involved three checks to identify long methods, each requiring detailed analysis and careful implementation to ensure code quality and maintainability. The complexity and depth of this task were underestimated in the initial planning, resulting in an extended duration for its completion. This delay was visible in the burn-down chart for Sprint 3, where the number of remaining tasks plateaued before resuming a downward trend towards the end of the sprint. This delay affected the velocity temporarily, causing a slight reduction in the number of completed story points for that sprint.

Despite the delay in Sprint 3, subsequent sprints saw a consistent application of the original planning process, with an increased focus on maintaining productivity. This is reflected in the improved and consistent velocity observed from Sprint 4 onwards. It was possible to catch up on the backlog and maintain a steady pace of progress, ensuring the successful completion of the planned story points by the end of the project.

Overall, the project execution demonstrated a positive trend in terms of speed and velocity. The initial sprints established a robust foundation with a high velocity, and despite the mid-project delay, I was able to effectively recover and maintain a steady pace to finish the project on time. The utilisation of the Agile methodology facilitated regular feedback and continuous improvement, allowing for adaptation to challenges and optimisation of workflow.

The execution of the master thesis project was characterised by a structured and iterative approach, leveraging the Agile methodology to ensure continuous progress and adaptability. The project achieved a high completion rate, with a total of 104 successfully completed tasks. The average velocity provided a stable measure of capacity and efficiency.

6 Implementation

6.1 DevOps Practices

6.1.1 Version Control and CI/CD Practices

In modern software development, version control systems (VCS) [33] are essential for managing code changes, facilitating collaboration, and ensuring a stable development workflow. Git, a distributed version control system, enables developers to accurately track and efficiently manage changes to the code base. GitHub, a web-based platform that leverages Git, further enhances collaboration by providing a suite of tools for hosting Git repositories, tracking issues, reviewing code, and more.

In the context of the current project, Git [34] and GitHub [35] serve as the foundational tools for source code management. Extensive use of branches ensures the stability of the mainline while allowing features to be developed, bugs to be fixed and new ideas to be explored.

The branching strategy uses two main types of branches: the main branch and feature branches. The main branch is the cornerstone of the codebase, representing the latest production-ready version of the software. It is the source of truth for the software development and deployment lifecycle. On the other hand, feature branches are derived from the main branch for the purpose of developing new functionality or fixing bugs. These branches are temporary, designed to be merged back into the main branch once the development work has been satisfactorily completed. This methodology facilitates a streamlined and efficient workflow and ensures that the main branch maintains its integrity as the central hub of stable, deployable software.

6.1.2 Continuous Integration and Continuous Deployment (CI/CD)

CI/CD [36] practices are fundamental components of the development workflow, emphasising the importance of automation in maintaining software quality and accelerating delivery. Continuous Integration (CI) ensures that the code base is automatically built and tested with each commit, keeping the software in a constantly releasable state. Continuous Deployment (CD) extends this process by automatically deploying software to production after it has successfully passed through the CI pipeline, facilitating the rapid delivery of features and fixes to users.

A bespoke CI/CD pipeline, using GitHub Actions [37], has been designed (shown on Figure 6.1) to integrate seamlessly with the project's GitHub repository. This pipeline, called 'Pipeline', is activated by workflow_dispatch, pull_request and push events targeting the main branch and any branch prefixed with releases/*. Such a configuration guarantees comprehensive coverage of all code changes, including both direct commits and contributions via pull requests.

The pipeline has three primary jobs:

1. *Build job.* This job initialises the Java development environment using JDK 17 and compiles the project using Gradle [38]. To improve efficiency, it uses caching mechanisms and reuses previously downloaded dependencies to speed up the build process.
2. *Qodana Job.* Following the successful completion of the build job, this phase performs a thorough code quality analysis using Qodana [39], a sophisticated code inspection tool developed by JetBrains. It scrutinises the codebase for potential inconsistencies and quality issues, ensuring that high standards of code quality are maintained.

3. *Dependency submission job.* This final task is to submit a comprehensive dependency graph of the project to GitHub. This submission facilitates the activation of Dependabot Alerts, which play a critical role in vulnerability scanning and dependency update management, thereby strengthening the security and reliability of the software.

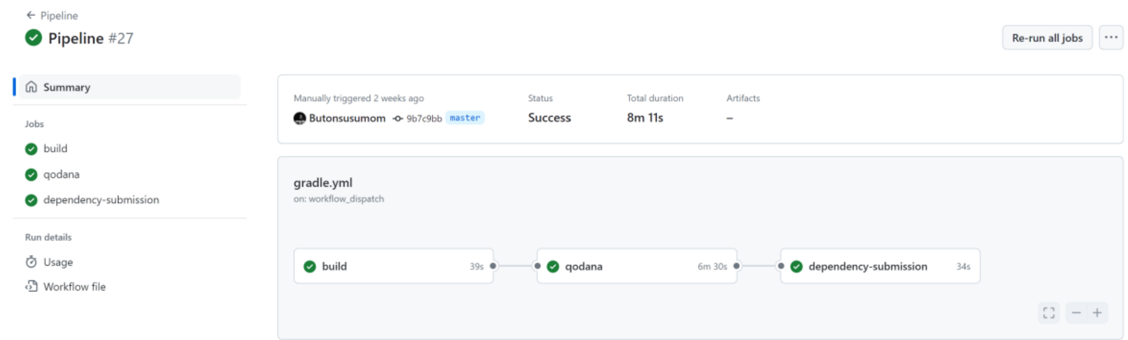


Figure 6.1 – Project pipeline

This strategic implementation of CI/CD via GitHub Actions underscores a commitment to quality, security and efficiency, streamlining the development cycle and ensuring that the software remains robust, secure and agile to meet the needs of users and developers alike.

6.1.3 The Role of Qodana in Ensuring Code Quality

The integration of Qodana [39], a code analysis tool developed by JetBrains, into the project's CI/CD pipeline, facilitated by GitHub Actions, represents a significant step forward in improving the quality and security of the codebase. This integration introduces an automated review mechanism into the development workflow, allowing issues to be identified and addressed at an early stage, improving the reliability and maintainability of the code. Qodana analyses of master branch is shown on Figure 6.2.

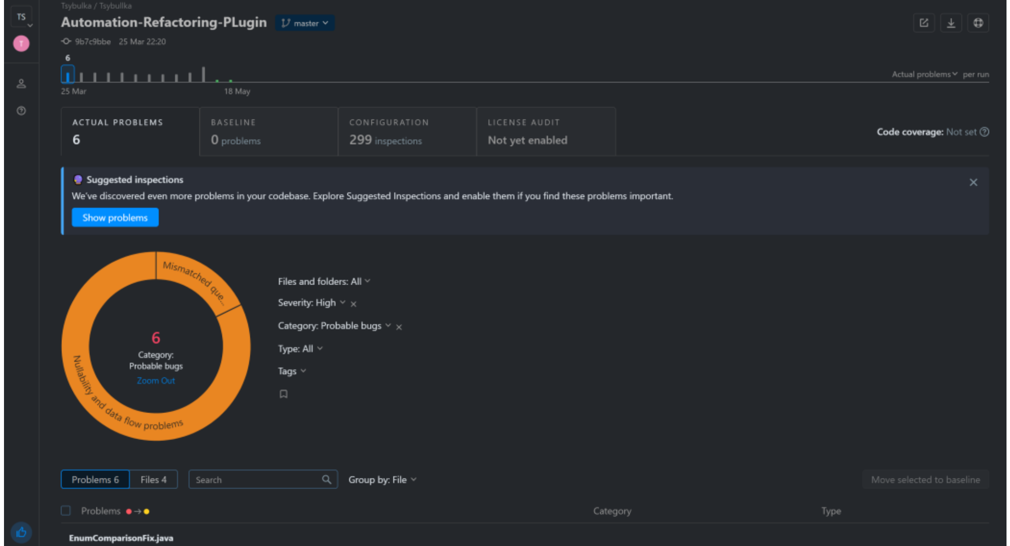


Figure 6.2 – Qodana report

Integration Highlights:

1. *Automated code quality checks.* Embedding Qodana into the GitHub Actions pipeline automates code quality and security checks, ensuring that every commit and pull request is checked for potential defects before being merged into the main branch.

2. *Customise for project needs.* Adjusting Qodana's auditing profiles to a project's unique requirements allows for focused auditing of the most important and pressing issues. This level of customisation ensures that development efforts strictly adhere to pre-defined quality standards and objectives.

Realised benefits:

1. *Improved code quality.* The proactive identification of bugs, code smells and vulnerabilities facilitated by Qodana results in a cleaner, more robust codebase. This proactive approach ensures fewer problems in production, resulting in a more reliable product for end users.
2. *Increased code coverage.* Incorporating code coverage analysis into CI/CD pipeline highlights areas of code that are under-tested or not tested at all. This visibility leads to increased test coverage, ensuring that new and existing features are thoroughly validated. High test coverage is synonymous with high code quality. It indicates a comprehensive assessment of software functionality through testing.
3. *Confidence in code integrity.* Integrating Qodana's detailed code analysis significantly increases confidence in the integrity and reliability of the codebase. It allows developers and stakeholders to be confident that the software adheres to high quality standards and has been thoroughly tested for security. This comprehensive analysis ensures that any issues are identified and addressed, resulting in a robust and secure software product.

Incorporating code coverage analysis, along with Qodana's advanced code inspection mechanisms, into a CI/CD infrastructure represents a dedicated commitment to maintaining the highest standards of code quality and security. This approach gives the ability to deliver functionality, underpinned by the confidence that the code base has been meticulously reviewed and rigorously tested.

6.2 Implemented code

The implementation phase of this project was crucial in translating the theoretical frameworks, methodologies and algorithms presented in earlier chapters into a working system. During this phase, a total of 8 code inspections were implemented, each designed to check the code against a set of pre-defined quality standards and software development best practices.

In addition to the code inspections, 11 OOP metric calculations have been implemented to evaluate various aspects of the code's architecture and maintainability. The implementation of these metrics helps to identify potential areas for improvement and to ensure that the codebase remains robust and efficient. Code is provided as a link to public repository at Annex 11.1.

The implementation involved a significant amount of coding. In total, the project comprises 7,543 lines of code organised into 65 files (java classes, kts, XML and HTML files). This large amount of code highlights the complexity and scale of the project. The organisation of the code into a significant number of classes indicates a commitment to maintaining a modular and scalable architecture.

On completion of the initial coding phase, the Qodana [39] code analysis tool was used to perform a thorough evaluation of the code base. And initially, Qodana identified 20 potential problems within the code (Figure 6.3). These

issues ranged from optimisations to critical vulnerabilities that required immediate attention.

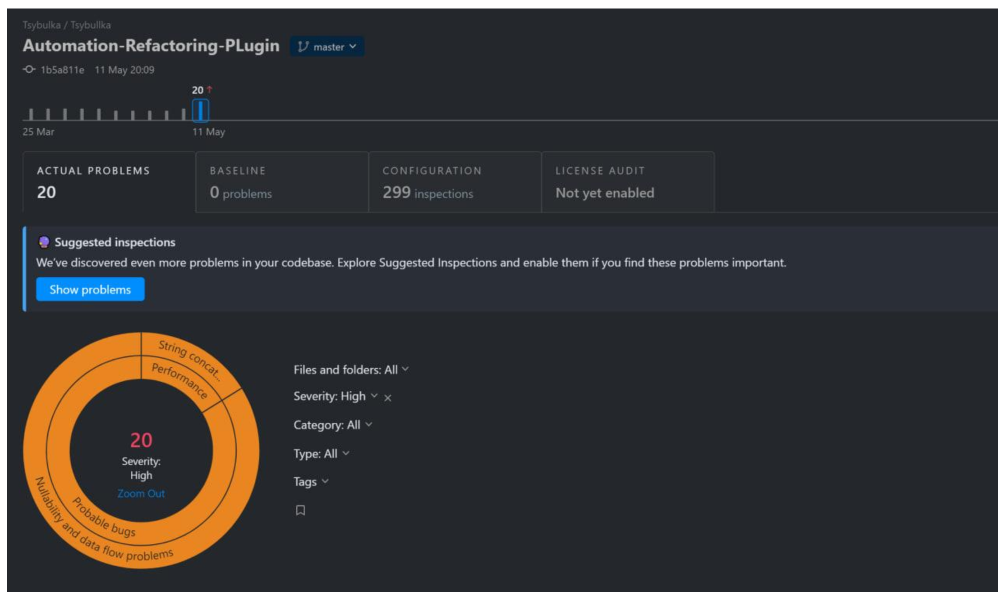


Figure 6.3 – Qodana report after initial development phase

Following the initial analysis, a process of code refinement and cleanup was undertaken. The objective of this phase was to address all identified issues systematically and enhance the overall quality of the software. This process involved revising code segments, optimising performance, and ensuring compliance with coding standards.

After the cleanup process, a final round of code analysis was conducted using the same Qodana tool. The outcomes of this analysis were significantly positive, as no problems with the source code were detected (Figure 6.4).

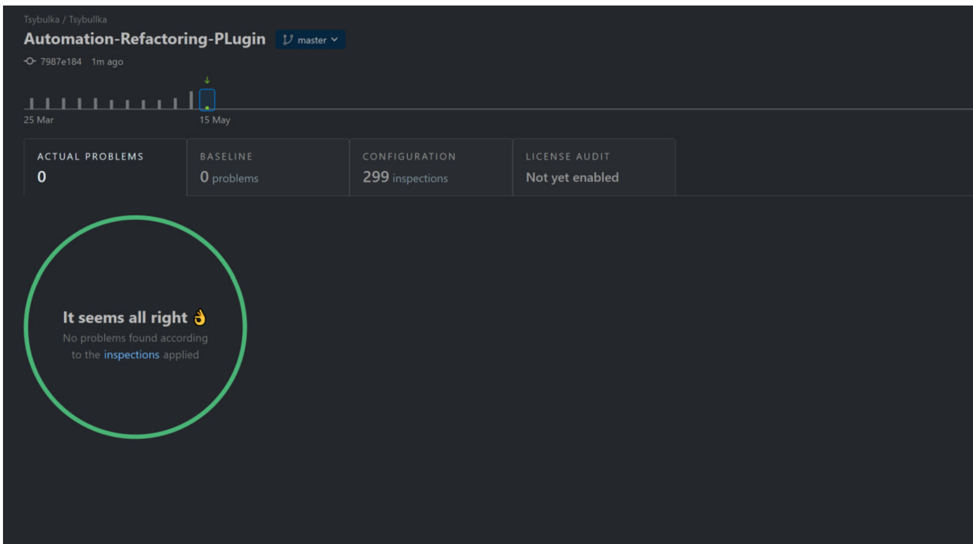


Figure 6.4 – Qodana report after clean up precess

This zero-defect status after the cleanup serves to validate the effectiveness of the remedial measures taken, while simultaneously underscoring the robustness of the codebase in question. Furthermore, the final continuous integration pipeline, described in the previous chapter, was consistently green, thus providing evidence of the reliability and stability of the project (Figure 6.5).

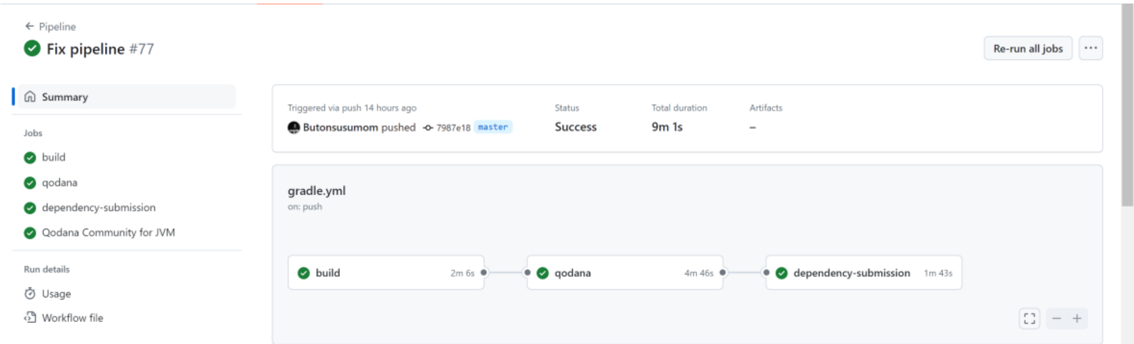


Figure 6.5 – Project pipeline after finalizing implementation

7 Testing and Evaluation

To ensure the reliability and functionality of the code, testing is a critical aspect of software development. Unit testing focuses on examining individual code modules in isolation, while integration testing evaluates the interaction between different components within a system. Manual testing, on the other hand, provides a human perspective and uncovers issues that automated testing may miss.

7.1 Unit Testing

Unit tests are a way of testing individual code modules (typically a single function or class in the case of object-oriented code) in an isolated environment. This means that if the code relies on external classes, placeholder classes are substituted instead. The code should not interact with networks (or external servers), files, databases (otherwise, the test would not only be testing the function or class itself but also the disk, database, etc.).

Using the Arrange-Act-Assert (AAA) pattern when writing unit tests significantly improves the chances of other developers understanding your code. This pattern simply divides and groups the test code into three sections, providing a readable structure for the unit test:

1. *Arrange*. Setting up initial conditions.
2. *Act*. Executing the functionality being tested.
3. *Assert*. Comparing expected values with the obtained ones.

An example unit test is provided in Figure 7.1.

```
@Test
public void shouldReturnFilePath_whenGetFilePath_getValidPsiFile() {
	// given
	String expectedFilePath = "/path/to/file";
	when(psiClass.getContainingFile()).thenReturn(psiFile);
	when(psiFile.getVirtualFile()).thenReturn(virtualFile);
	when(virtualFile.getPath()).thenReturn(expectedFilePath);

	// when
	String actualFilePath = classUnderTest.getFilePath(psiClass);

	// then
	assertEquals(expectedFilePath, actualFilePath);
}
```

Figure 7.1 - Example of unit test implementation

The use of additional tools such as JUnit 5 and Mockito is essential for implementing effective unit tests in Java, as they offer a rich set of features and mocking capabilities to ensure the robustness and reliability of code, especially in scenarios such as developing a refactoring plugin, where thorough testing is critical. JUnit 5 [40] is the latest version of the popular unit testing framework for Java. It provides a rich set of features for writing and executing unit tests effectively. Mockito [41] is a mocking framework that allows developers to create mock objects in tests, making it easier to isolate the code under test from its dependencies.

The core functions of code refactoring were tested using unit tests. To measure code coverage the internal system of IntelliJ IDEA was initially used. In the next phase of development, it is planned to integrate Qodana code

coverage analyses tool, a more accurate and comprehensive code coverage tool, to enhance the development process.

The threshold for test coverage was set at 70%. This threshold strikes a balance between ensuring sufficient coverage to catch most defects while recognizing the practical constraints of achieving higher coverage levels. At 70%, the coverage is high enough to provide confidence in the software's reliability and maintainability without overburdening the development process with diminishing returns on additional tests.

In total, 120 tests were implemented, covering 1557 lines out of 2177 lines of source code, achieving 71% coverage. It is important to note that resource files, configuration files for dialogues (e.g., XML, HTML), and other non-source code files were excluded from this coverage metric. Code coverage report is presented on Figure 7.2.

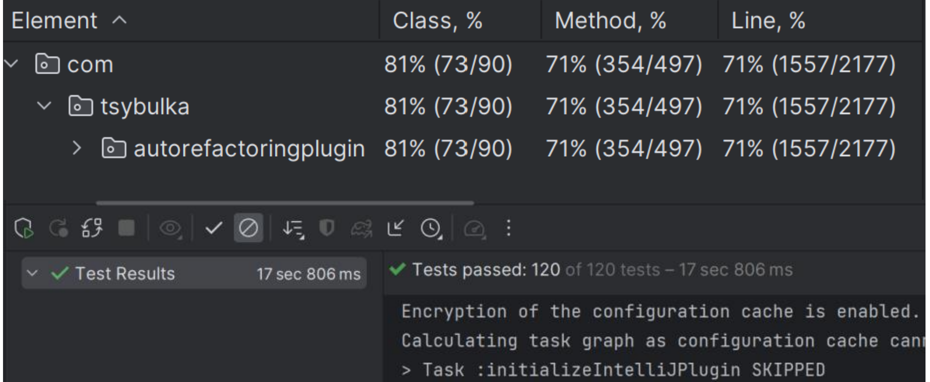


Figure 7.2 – Code coverage report

All tests are maintained on a separate branch of the repository, named `testing_process`, which helps in managing and organizing the testing code separately from the main development branch. Access to the branch with tests can be found in Annex 11.2.

The process of writing unit tests for an automation refactoring plugin presented a number of challenges:

1. *The PSI interface*. The Program Structure Interface (PSI) in IntelliJ IDEA is a sophisticated tool for code analysis and manipulation. However, it was also complex and required a profound understanding to be employed effectively in unit tests.
2. *The extensive use of mocks and static mocks*. The plugin was dependent upon IntelliJ's Application Programming Interfaces (APIs) and other external systems. In order to isolate the unit tests, it was necessary to make extensive use of mocks and static mocks. This added a further layer of complexity, as the mocks had to be configured with great precision in order to replicate the behaviour of the real objects.
3. *Dynamic and state-dependent behaviour*. The necessity for refactoring operations to be contingent upon the current state of the codebase, which could undergo dynamic changes, was a further complicating factor. The writing of unit tests that accounted for this dynamic behaviour required careful planning and consideration.
4. *Integration with IntelliJ IDEA*. The plugin was designed to integrate deeply with the IntelliJ IDEA platform, relying on its application programming interfaces (APIs) and internal mechanisms. The challenge of

accurately simulating the IntelliJ environment in the context of unit testing was significant.

The unit testing of the automation refactoring plugin in IntelliJ IDEA proved to be a challenging endeavour due to the intricate PSI interface, the extensive use of mocks, and the dynamic code behaviours. Despite the aforementioned challenges, a robust test suite was implemented, thereby achieving a high level of code quality. In the future, the implementation of additional tests, improvements to code coverage, and the integration of Qodana will enhance the accuracy of test coverage, thereby facilitating continuous improvement of the plugin.

7.2 Integration Testing

Integration testing is about examining the interaction between different modules or components within a system. Its basic purpose is to validate the combined functionality of these components. In the context of a refactoring plugin, integration testing is of paramount importance as it validates the seamless interaction of the plugin within the IntelliJ IDEA environment. It also verifies that the plugin interacts harmoniously with other IDE components and external dependencies.

The primary objective of integration testing within a refactoring plugin is to determine the accuracy and effectiveness of the code smell detection algorithms. This involves creating various “happy case” code scenarios to validate the plugin's ability to accurately identify potential problems. By exposing the plugin to various simulated scenarios, its reliability in effectively detecting code smells can be ensured.

Integration testing within the IDE environment aims to comprehensively replicate real-world code scenarios. This involves creating test cases with code snippets containing predefined code smells, and then evaluating the plugin's detection capabilities. The complexity of the test cases can vary to cover a wide range of potential code issues, ensuring thorough validation of the plugin's functionality.

For each code inspection supported by the plugin, specific code snippets were created to represent potential code smells. Integration tests were then developed to simulate these scenarios within the IntelliJ IDEA environment, example presented on Figure 7.3. This approach facilitated rigorous evaluation of the plugin's ability to accurately detect and resolve various code issues, thereby increasing its reliability and effectiveness in real-world development scenarios.

For a typical integration test, the process involves the following steps:

1. *File consumption.* The test initially loads a code file that contains pre-written scenarios simulating "real-world" code structures with potential issues, known as code smells.
2. *Problem identification.* Once the file is loaded, the plugin under test is activated to scan and identify problems using the specified code inspection tools integrated into the plugin.
3. *Quick fix application.* If the plugin identifies an issue and suggests a quick fix, the test automatically applies this fix to the code. This step tests the plugin's ability to not only recognize problems but also to suggest and implement automatic corrections.

4. *Validation*. After applying the quick fixes, the test compares the modified code against a predefined expected result. This comparison checks whether the applied fixes correctly address the issues without introducing new ones, confirming the effectiveness of the plugin.

```
@Test
public void testObjectComparisonFix() {
	myFixture.configureByFile("before.java");
	InspectionProfileEntry inspection = new ObjectComparisonInspection();
	myFixture.enableInspections(inspection);

	List<HighlightInfo> highlightInfos = myFixture.doHighlighting();
	List<IntentionAction> quickFixes = myFixture.getAllQuickFixes();
	for (IntentionAction quickFix : quickFixes) {
		if (quickFix.getText().equals(QUICK_FIX_NAME)) {
			myFixture.launchAction(quickFix);
			break;
		}
	}
	myFixture.checkResultByFile("after.java");
}
```

Figure 7.3 – Example of integration test implementation

However, it was not possible to implement this process for all code inspections because some quick fixes require user decisions through refactoring dialogues (long method, cyclomatic complexity, test naming convention and scattered functionality code inspections). These scenarios involve more complex refactoring operations where automated testing might not effectively replicate a real user's decision-making process. For these specific code inspections, the integration tests are designed differently:

1. *Highlighting and detection*. Instead of applying fixes, tests for these inspections focus on ensuring that the plugin correctly highlights the problematic code elements. Each highlight is accompanied by a message that informs the user of the nature of the detected issue.
2. *User interaction requirement*. The tests acknowledge the need for user interaction to resolve the highlighted issues. In these cases, rather than applying a fix, the test ensures that the necessary user input options are presented correctly, mimicking the user's decision-making process in the real-world scenario.
3. *Partial automation*. While the full cycle of detection, fixing, and validation cannot be automated for these inspections, ensuring that issues are correctly identified and highlighted represents a critical test of the plugin's ability to assist in the refactoring process.

This approach to integration testing balances the need for automated testing where possible while recognizing and planning for scenarios that require human judgment. This ensures that the plugin not only works efficiently in a typical development environment but is also robust enough to handle complex refactoring tasks that necessitate developer input.

In total, 8 integration tests were implemented to validate the functionality of the refactoring plugin, ensuring comprehensive coverage of its features. All integration tests were placed on the same branch as the unit tests, named `testing_process`. Access to the repository can be found in Annex 11.2.

7.3 Manual Testing

The importance of manual testing cannot be overstated, as it provides a human perspective that automated testing often lacks. While automated tests are invaluable for their efficiency and repeatability, they may overlook certain aspects that only a human tester can detect, such as usability issues, aesthetic concerns, or unexpected user interactions.

7.3.1 Functional requirements testing

For the validation of the developed plugin's functionality, a series of test cases were devised, the descriptions of which are outlined at Annex 11.3. These test cases involve executing specific scenarios while continuously verifying the obtained results. Each test case is meticulously crafted to encompass multiple components within a single test scenario. The successful execution of these test scenarios ensures the accurate behavior of the plugin within the IntelliJ IDEA environment.

In conclusion, the functional requirements testing of the developed plugin involved a total of 32 test cases, aimed at evaluating the plugin's performance within the IntelliJ IDEA environment. Out of these, 30 test cases successfully passed, demonstrating the robustness and accuracy of the plugin in handling settings validation, and computing key software metrics. However, there were two failed test cases, specifically related to the inspections for checking method length and repeated object creation. These failures indicate areas that require further attention and refinement to meet the desired standards of functionality. Moving forward, addressing these issues was crucial to enhancing the overall reliability and effectiveness of the plugin in real-world development scenarios.

7.3.2 Nonfunctional requirements testing

In addition to functional requirements, it is essential to validate non-functional requirements to ensure the overall quality and usability of the plugin. Non-functional requirements often focus on aspects such as performance, compatibility, memory usage, and security, which are crucial for a seamless user experience. Annex 11.4 details the non-functional requirements tested for the plugin, along with their actual results and test outcomes. A total of 8 nonfunctional test cases were implemented to evaluate the plugin's performance, compatibility, memory usage, and security, and all 8 tests passed successfully.

The non-functional testing of the plugin showed excellent performance, compatibility, and compliance, confirming its effectiveness and reliability in various operational environments. It met key performance benchmarks, demonstrated seamless interoperability with existing tools, and adhered to stringent data privacy and resource efficiency standards. This comprehensive validation underscores the plugin's readiness for deployment and its capability to enhance user productivity in diverse development scenarios.

7.4 Beta Testers Review

7.4.1 Overview

In software development, thorough testing by experienced users is crucial for several reasons:

1. *Identify hidden bugs*. Users with a deep understanding of the technology often uncover problems that may not be obvious to developers or automated tests.
2. *Provide user-centric design feedback*. Testers provide valuable insights that can improve the usability and functionality of software from a user's perspective.
3. *Validate functionality*. Testing ensures that the software does what it is supposed to do and works under different conditions, confirming its reliability and effectiveness.

The test group consisted of 3 Java developers, each with more than two years of experience and they were using the plugin for a week. They were selected from different companies and different projects to ensure a wide range of insights. This diversity was needed because of:

1. *Diversity of experience*. Different backgrounds bring unique challenges and use cases, providing a comprehensive assessment of the plugin's capabilities.
2. *Cross-industry validation*. Feedback from different sectors tests the plugin's adaptability and usefulness in different contexts, which is particularly important in environments with specific requirements, such as healthcare or finance.

Overall, testers reported a positive experience with the plugin. They found it particularly useful for analysing projects and gaining insights that facilitated code improvements. The ability to visualise and effectively address specific code smells made the plugin a valuable addition to their development toolkit.

7.4.2 Positive Feedback

The project analyses panel (Figure 7.4) received high praise for its ability to provide a comprehensive view of the overall health of projects. It effectively categorizes code smells, making it straightforward for users to identify and prioritize issues based on their type, such as test-related, architectural, or implementation flaws. This feature greatly simplifies the tracking and addressing of specific problems, thus enhancing the efficiency of project maintenance.

The customization options available in the plugin were highly valued, particularly in specialized fields where standard practices may vary significantly. For instance, the ability to disable inspections for "Long methods" in domains like medicine, where complex methods are necessary, was seen as crucial. This level of customization ensures that the tool can be adapted to a wide range of coding standards and practices, making it more useful across different projects.

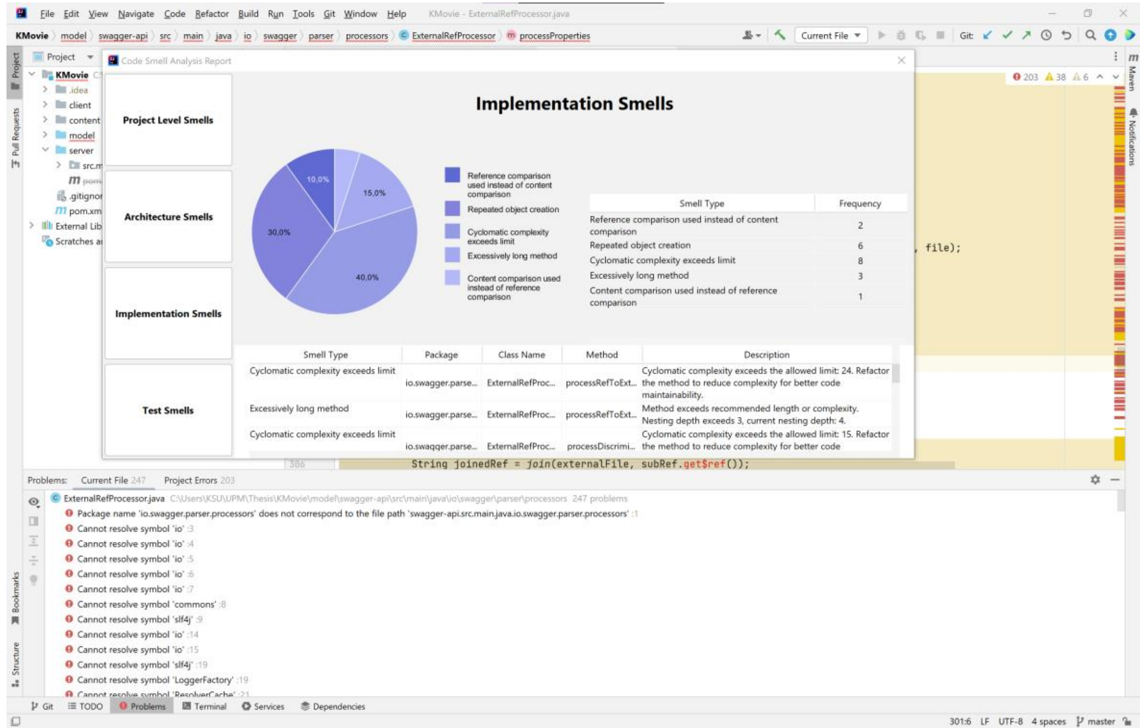


Figure 7.4 – Project analyses result of beta tester

The detailed explanations provided for each detected code smell, example shown on Figure 7.5, were found to be extremely beneficial. These explanations help developers understand the rationale behind each flag, promoting a deeper understanding and encouraging better coding practices. This feature is essential for helping developers learn and apply best practices in their daily programming tasks, thereby improving code quality over time.

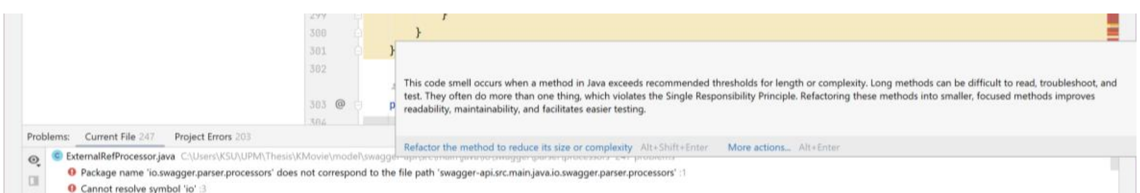


Figure 7.5 – Explanation for detected code smell

7.4.3 Places for improvements

Feedback indicated that the layout of the project analysis panel could not accommodate different screen sizes or user preferences, which limited its usability (Figure 7.6). Improving this aspect of the plugin would enhance the user experience by making it more adaptable and easier to use on a variety of devices and configurations.

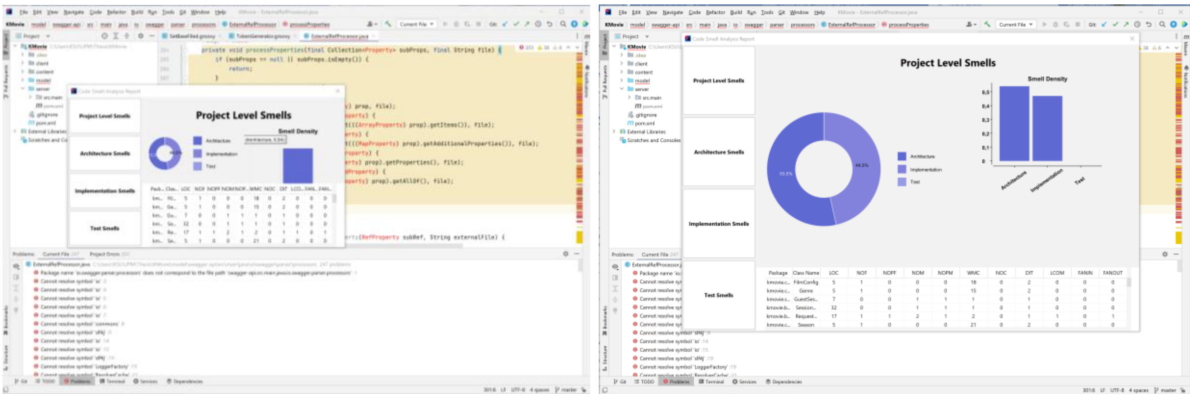


Figure 7.6 – “Project analyses” panel when resizing

Testers suggested integrating hyperlinks directly into the project analysis table to improve navigation efficiency. This improvement would allow users to click on class or method names and be taken directly to the specific location of the problem within the codebase. Such a feature would streamline the refactoring process by reducing the time spent manually searching through files. This and the previous suggested improvements would be implemented in the next release.

A significant bug was identified in the refactoring functionality, specifically related to the "Object Method Parameter" code smell. When method parameters are modified to fix this smell, the corresponding method calls in the code are not automatically updated, leading to potential errors and inconsistencies, Figure 7.7. Fixing this bug is critical to implement in current release, ensuring that the plugin's refactoring actions are reliable and effective, and that the integrity of the codebase is maintained after refactoring.

```
2 usages
public String method3(TestObject s1) { return s1.field3; }

public void testMethodCall(TestObject s1) {
    method1(s1);
    method3(s1);
}
```

```
2 usages   1 related problem
public String method3(String field3) { return field3; }

public void testMethodCall(TestObject s1) {
    method1(s1);
    method3(s1);
}
```

Figure 7.7 – "Object Method Parameter" quick fix bug

7.5 Final Evaluation

During the final stages of testing and evaluation of the software, several issues were identified and resolved to ensure the robustness and reliability of the system. This chapter describes the bugs found during manual and beta testing, the steps taken to fix them, and the subsequent verification through unit, integration and manual testing.

During manual testing, 2 functional bugs, described in Chapter 7.3.1, were identified that required immediate attention before the first release. The identified issues were addressed, with fixes implemented and thoroughly retested to confirm their resolution. This proactive approach ensured that the first release met the expected standards of functionality and user satisfaction.

Beta testing provided valuable insights from real user environments and identified 3 key areas of concern that required refinement:

1. *"Project analyses" panel stability and responsiveness*. Initially, the UI panes were flexible, but this led to inconsistencies in the user experience across different screen sizes and resolutions. To address this, the UI panes have been made fixed and non-resizable, improving the stability and appearance of the application across devices. There are plans to upgrade the UI to adaptive designs in future releases to improve accessibility and user engagement.
2. *Hyperlinks for methods and classes*. Another issue identified was the lack of hyperlinks for methods and classes within the documentation, which hindered the navigation and usability of the help sections. This would be rectified in the next release cycle by embedding hyperlinks, thus improving the interactivity and usefulness of the documentation for users.
3. *Functional bug*. There was also 1 functional bug found related to one of the code inspection quick fixes. This has been addressed in the current release.

Once the fixes had been implemented, a rigorous testing protocol was followed:

1. *Unit and integration testing*. All unit and integration tests were re-run and the software passed successfully, indicating that the fixes had not introduced any new problems and that the existing functionality was intact and working as expected.
2. *Manual test cases reviews*. All manual test cases were rechecked to ensure comprehensive test coverage. This reassessment helped to confirm that all identified issues had been adequately addressed and that there were no residual effects that would affect the stability or performance of the system.

The thorough testing and fixing of the identified bugs significantly improved the quality of the software. The fixes not only addressed immediate functional and usability issues, but also laid the groundwork for future enhancements, contributing to a robust and user-friendly application. This final evaluation phase was critical in ensuring that the software met the high standards required for release, paving the way for subsequent updates and enhancements.

8 User Documentation

As the primary documentation creation tool the AsciiDoc plugin for IntelliJ IDEA was used [42]. This plugin allows for seamless integration into the development environment, facilitating the creation of documentation alongside code development. AsciiDoc is a text document format for writing notes, documentation, articles, books, e-books, slide shows, web pages and blogs. The AsciiDoc format is simple yet powerful, supporting complex documentation needs such as diagrams, mathematical notation, and bibliographies.

The AsciiDoc plugin provides a live preview feature (Figure 8.1) that is used extensively during the documentation creation process. This feature allows the developer to view the rendered HTML version of the documentation in real time, ensuring that formatting and content flow are maintained as intended.

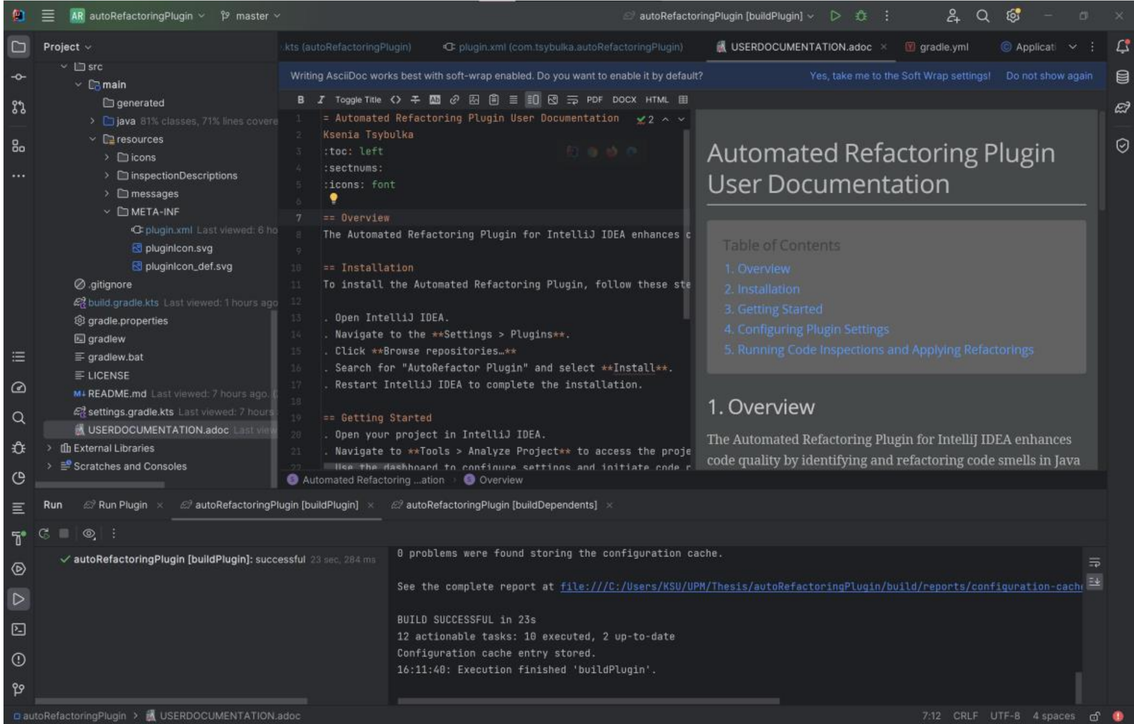


Figure 8.1 – AsciiDoc plugin usage

The final step in the documentation process is to publish the documentation using GitHub Pages. This service enables the conversion of AsciiDoc files to HTML and hosts the resulting content directly from a GitHub repository. The process involves setting up a GitHub Pages site linked to the repository containing the AsciiDoc files. Automated tools within GitHub Pages regenerate the HTML content each time updates are pushed to the repository, ensuring that the documentation is always up to date. A new GitHub action is run each time something is pushed to the master branch to update the documentation (Figure 8.2).

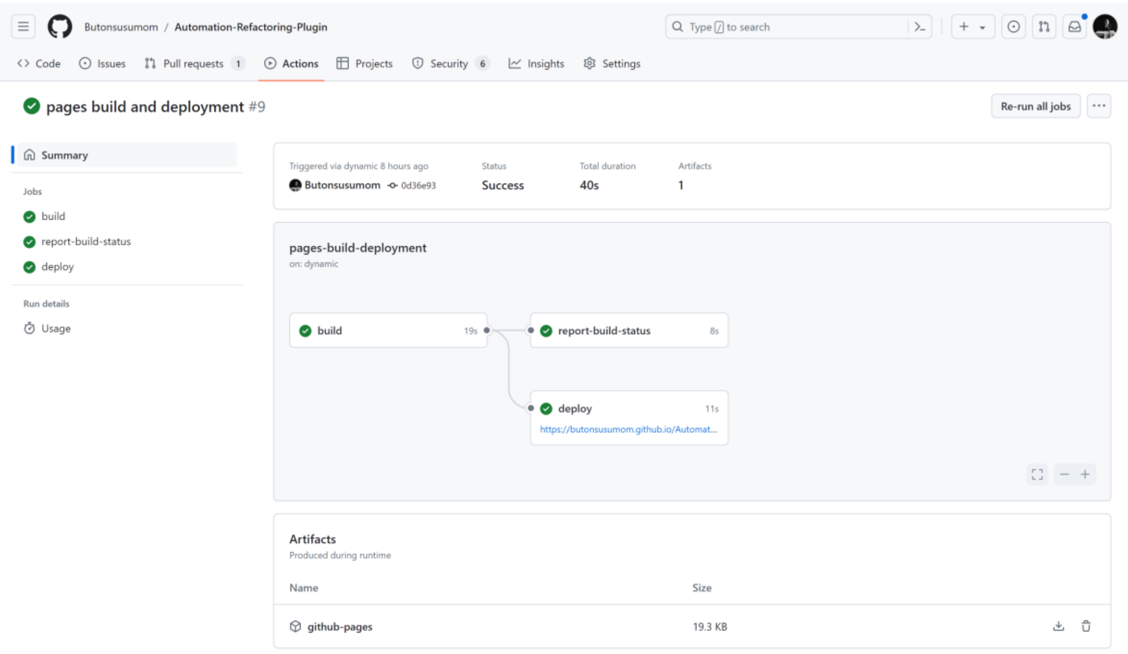


Figure 8.2 – GitHub action for documentation deployment

Once published, the documentation is accessible to users via a dedicated URL (Annex 11.5) provided by GitHub Pages. This approach not only makes the documentation readily available to users, but also simplifies ongoing maintenance and updates. As updates and enhancements are made to the plugin, corresponding updates to the documentation can be published quickly, keeping the user community well informed. The final documentation placed on the URL is shown on Figure 8.3.

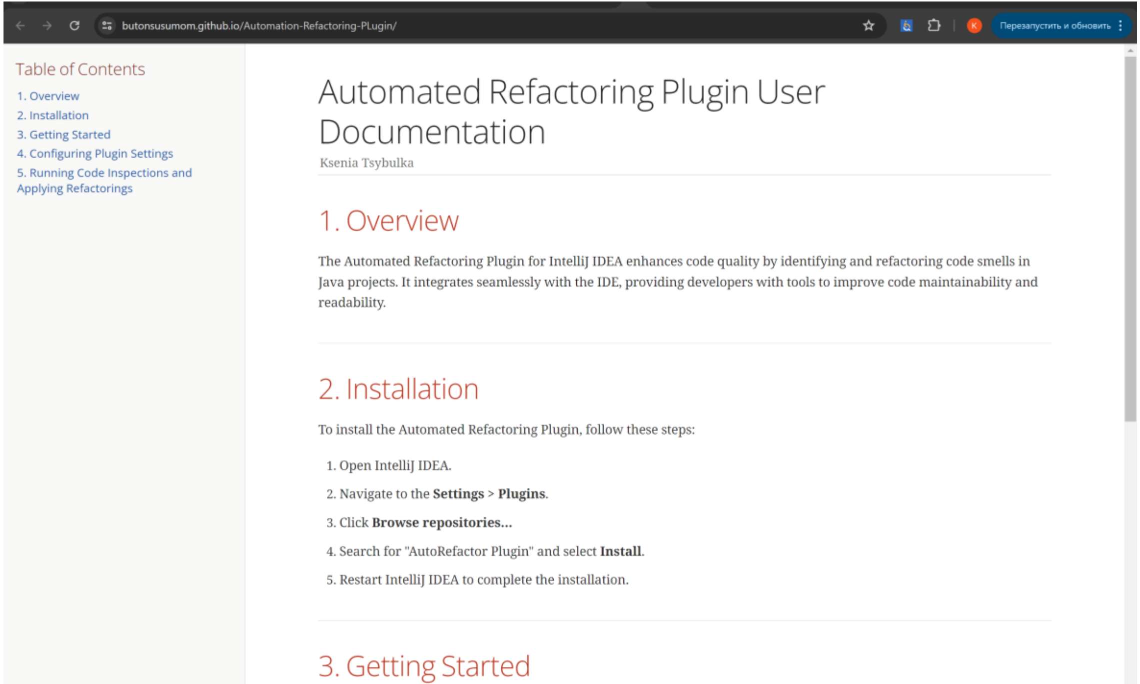


Figure 8.3 – Plugin user documentation deployed by GitHub Pages

9 Conclusions and Future Work

This thesis has addressed the need for advanced automated refactoring tools within the software development community, culminating in the design and implementation of a novel plugin for IntelliJ IDEA. This project was grounded in a comprehensive analysis of the current academic and practical landscape of automated refactoring, which identified key areas where existing tools fell short in supporting modern development practices.

Through a review of literature and existing technologies, this research synthesized a deep understanding of automated refactoring principles and their application. The gaps identified, particularly in terms of usability, functionality, and integration with existing workflows, informed the development of a plugin that not only addresses these issues but also enhances code quality, maintainability, and readability.

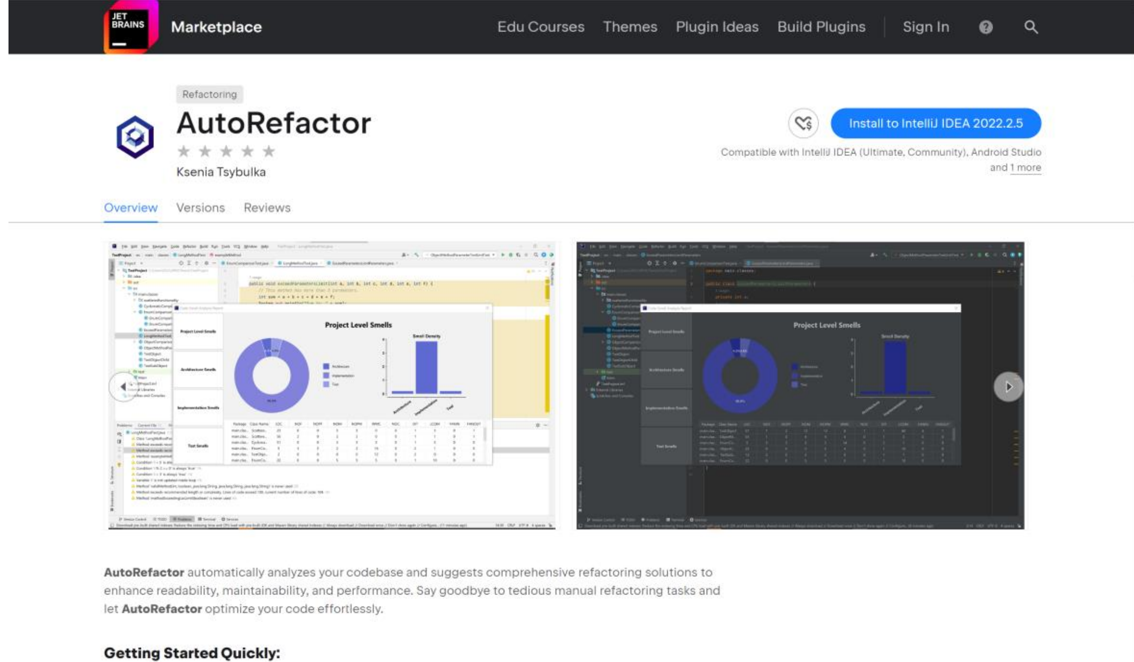


Figure 9.1 – Plugin download page on JetBrains Marketplace

The newly developed plugin, which was created as a direct result of this thesis, was rigorously tested, and subsequently made available for free via the JetBrains Marketplace [43], Figure 9.1. The link for plugin download is located at Annex 11.6. The availability of this plugin on a popular platform ensures it reaches a broad audience, thereby promoting widespread use and ongoing enhancement driven by community feedback. Furthermore, this approach allows for the sustainability of the plugin through optional donations via buymeacoffee [44] donation service, providing a means for users to support further development and refinement (Figure 9.2).

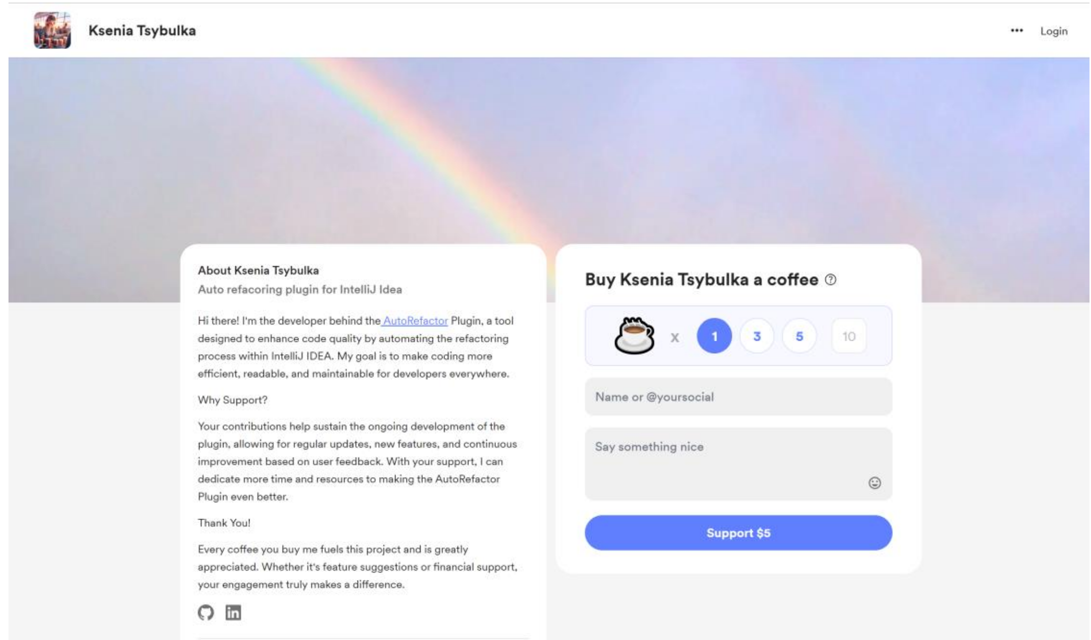


Figure 9.2 – Donation for plugin development page

As of 4 June 2024, the plugin had achieved notable adoption, evidenced by a significant number of downloads, which reflects the software development community's endorsement of the tool's value. Downloads analyses shown on Figure 9.3.

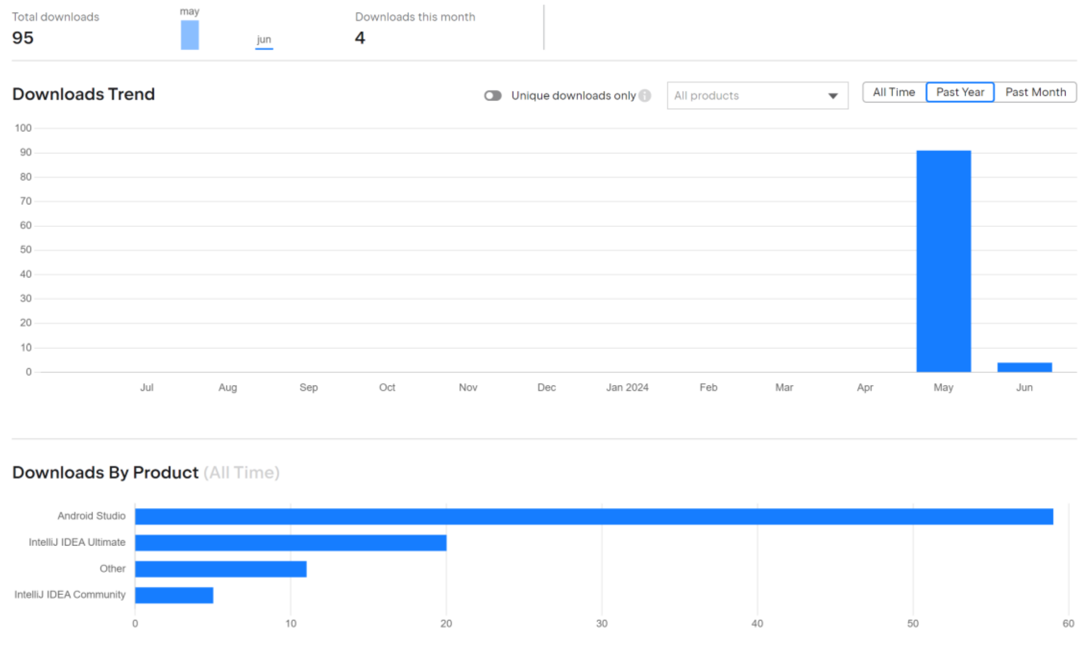


Figure 9.3 – plugin downloads report

The successful initial release of the IntelliJ IDEA refactoring plugin marks only the beginning of its development lifecycle. As the project progresses, several enhancements are planned to refine its functionality, expand its capabilities, and better serve the software development community. The roadmap for future work is structured around specific improvements aimed at enhancing the plugin's robustness, usability, and effectiveness:

1. *Implementation of additional code inspections*. To build on the current capabilities, the next releases will include the implementation of 9 code inspections, outlined in Chapter 3.2.2, that were specified but not completed in the initial phase of project.

2. *Expanding functionality based on user feedback*. Continuous engagement with the plugin's user base will provide critical insights into the specific needs and challenges faced by developers using the tool. Based on this feedback, the development team will specify and implement additional code inspections tailored to the users' most pressing concerns. This iterative process ensures that the plugin remains responsive to the evolving demands of its community and keeps pace with the latest development practices.

3. *Refinement of current algorithms*. Specific algorithms showed limitations during initial testing highlighted in Chapters 4.6.2.1.1.2, 4.6.2.2.6.2, and 4.6.2.3.1.2. Future versions will focus on refining these algorithms to improve their accuracy and efficiency, thereby enhancing the plugin's overall effectiveness in automating refactoring tasks.

4. *User interface enhancements for "Project Analyses" panel.* Feedback on the Project Analyses panel has indicated a need for better display features. As detailed in Chapter 7.4.3, the interface will be improved, allowing users to easily interpret analysis results and take actionable steps.

5. *Improvement of code coverage.* Increasing the test coverage from 71% to at least 75% is a targeted goal intended to bolster the plugin's reliability. Enhanced code coverage ensures a broader range of functionalities is tested, reducing the likelihood of bugs and improving the stability of the plugin.

6. *Integration of integration tests run in CI/CD.* To maintain the integrity of the plugin through ongoing changes, integration tests will be added to the development pipeline. This step will verify that all code inspections operate effectively in "happy case" scenarios, ensuring consistent performance and reliability and that pushed code does not affect functionality.

7. *Performance optimization for large codebases*. The plugin will be stress-tested on extensive codebases exceeding 1,000 classes to identify performance bottlenecks. Subsequent optimizations will focus on improving processing speed and reducing memory usage, crucial for maintaining performance in large-scale software environments.

In conclusion, this thesis has not only bridged the gap between academic research and practical application but has also laid a solid foundation for ongoing enhancement of the IntelliJ IDEA refactoring plugin. The detailed roadmap for future development underscores a commitment to continuous improvement and adaptation to meet the evolving needs of the software development community. As the plugin matures, it promises to extend its utility and efficacy, underpinned by rigorous academic research and enriched by practical feedback from its users. The sustained development effort will further solidify the plugin's role in enhancing the adaptability and efficiency of software development processes, aligning with both current practices and future advancements.

10 Bibliography

[1] A. R. C. Martin, “Clean Code: A Handbook of Agile Software Craftsmanship”, Prentice Hall, 2008.

[2] B. C. Jones and O. Bonsignour, “The Economics of Software Quality”, Addison-Wesley, 2012.

[3]. I. Sommerville, “Software Engineering, 9th ed.”, Addison-Wesley, 2011.

[4] D. R. S. Pressman, “Software Engineering: A Practitioner's Approach”, McGraw-Hill, 2014.

[5] F. J. Nielsen, “Usability Engineering”, Morgan Kaufmann, 1993.

[6] G. H. Li and S. Henry, “Maintenance metrics for the object-oriented paradigm”, Proceedings of the 15th International Conference on Software Engineering, 1993.

[7] H. L. C. Briand, J. W. Daly, and J. Wüst, “A Unified Framework for Cohesion Measurement in Object-Oriented Systems”, IEEE Transactions on Software Engineering, vol. 22, no. 11, pp. 1-17, 1996.

[8] I. S. R. Chidamber and C. F. Kemerer, “A Metrics Suite for Object Oriented Design”, IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493, 1994.

[9] J. J. E. Smith, “Automated Code Analysis: Advancements and Challenges”, IEEE Software, vol. 35, no. 4, pp. 92-98, 2018.

[10] K. M. Fowler, “Continuous Integration”, MartinFowler.com, 2006. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html. [Accessed: 01.03.2024].

[11] L. G. Robles, “Code Quality Metrics: A State-of-the-Art Review”, IEEE Software, vol. 37, no. 3, pp. 32-38, 2020.

[12] M. Refactoring Guru, “Refactoring and Design Patterns”, Refactoring Guru, [Online]. Available: https://refactoring.guru. [Accessed: 01.03.2024].

[13] M. M. Feathers, “Working Effectively with Legacy Code”, Prentice Hall, 2005.

[14] N. Kim, M., Cai, D., & Kim, S. “An Empirical Study of Software Refactoring for a Large-Scale System”, Proceedings of the International Conference on Software Engineering (ICSE).

[15] O. Murphy-Hill, E., Parnin, C., & Black, A.P. “How Do Developers Use Parallel Programming?”, Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.

[16] P. Ratzinger, J., Sigmund, T., & Vorburger, P. “Tool Support for Refactoring of Legacy Software Systems”, Proceedings of the International Workshop on Principles of Software Evolution.

[17] Q. Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. “Refactoring: Improving the Design of Existing Code”, Addison-Wesley Professional, 1999.

[18] A. Thompson, S., & Huang, G. “Mixed-Methods Study on Automated Refactoring's Impact on Open Source Software Quality”, Journal of Software Engineering Research and Development, vol. 10, no. 1, pp. 1-23, 2022.

[19] B. Sanchez, P., & Romero, F. “A Longitudinal Study on the Effects of Automated Refactoring in Software Maintenance”, Software Quality Journal, vol. 31, no. 2, pp. 445-467, 2023.

[20] C. Nguyen, A., et al. “Comparative Analysis of Automated Refactoring Adoption: An Empirical Study”, Proceedings of the 29th International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 113-124, 2021.

[21] D. Oliveira, J., & da Silva, T. “Experimental Evaluation of Automated Refactoring's Impact on Software Development Efficiency”, Empirical Software Engineering, vol. 25, no. 3, pp. 2034-2066, 2020.

[22] “IntelliJ IDEA” Wikipedia, The Free Encyclopedia. Wikimedia Foundation, Inc. [Online]. Available: https://en.wikipedia.org/wiki/IntelliJ_IDEA. [Accessed: 01.03.2024].

[23] “Hapag-Lloyd”. Hapag-Lloyd AG. [Online]. Available: https://www.hapag-lloyd.com/es/home.html. [Accessed: 02.03.2024].

[24] JetBrains. "Code Inspections". IntelliJ IDEA Plugin Development Kit. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/code-inspections.html. [Accessed: 02.03.2024].

[25] JetBrains. "ProblemsHolder.java". IntelliJ IDEA Community Edition. [Online]. Available: https://github.com/JetBrains/intellij-community/blob/master/platform/analysis-api/src/com/intellij/codeInspection/ProblemsHolder.java. [Accessed: 02.03.2024].

[26] Quick fix. Intellij web page. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/quick-fix.html?from=jetbrains.org. [Accessed: 02.03.2024].

[27] Program Structure Interface. Intellij web page. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/psi.html?from=jetbrains.org. [Accessed: 02.03.2024].

[28] General Threading Rules. Intellij web page. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/general-threading-rules.html?from=jetbrains.org#modalitystatenon-modal-nonmodal. [Accessed: 02.03.2024].

[29] McCabe, T. "A Complexity Measure." IEEE Transactions on Software Engineering, vol. SE-2, pp. 308-320, 1976.

[30] JetBrains, "Extract Method," [Online]. Available: https://www.jetbrains.com/help/idea/extract-method.html. [Accessed: 01.05.2024].

[31] Google Git, "CyclomaticComplexityVisitor," [Online]. Available: https://android.googlesource.com/platform/tools/idea/+/ec3fb1e06285c0467a7a20360ca80453bc7635d4/plugins/InspectionGadgets/InspectionGadgetsAnalysis/src/com/siyeh/ig/methodmetrics/CyclomaticComplexityVisitor.java. [Accessed: 01.05.2024].

[32] Notion. [Online]. Available: https://notion.soapps.me/. [Accessed: 14.03.2024].

[33] Wikipedia. Version Control System (VCS). [Online]. Available: https://en.wikipedia.org/wiki/Version_control. [Accessed: 01.04.2024].

[34] Git. [Online]. Available: https://git-scm.com/. [Accessed: 01.04.2024].

[35] GitHub. [Online]. Available: https://github.com/. [Accessed: 01.04.2024].

[36] RedHat. Continuous Integration/Continuous Deployment (CI/CD). [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd. [Accessed: 01.04.2024].

[37] GitHub. GitHub Actions. [Online]. Available: https://github.com/features/actions. [Accessed: 08.04.2024].

[38] Gradle. [Online]. Available: https://gradle.org/. [Accessed: 08.04.2024].

[39] JetBrains. Qodana. [Online]. Available: https://www.jetbrains.com/qodana/. [Accessed: 08.04.2024].

[40] JUnit 5. [Online]. Available: https://junit.org/junit5/. [Accessed: 12.04.2024].

[41] Mockito. [Online]. Available: https://site.mockito.org/. [Accessed: 12.04.2024].

[42] AsciiDoc Plugin. [Online]. Available: https://plugins.jetbrains.com/plugin/7391-asciidoc. [Accessed: 20.05.2024].

[43] JetBrains Marketplace. [Online]. Available: https://plugins.jetbrains.com/plugin/7391-asciidoc. [Accessed: 20.05.2024].

[44] Buy me a coffee donation service. [Online]. Available: https://buymeacoffee.com/. [Accessed: 22.05.2024].

11 Annexes

11.1 Source Code of Implemented Plugin

https://github.com/Butonsusumom/Automation-Refactoring-Plugin

11.2 Tests for Source Code

https://github.com/Butonsusumom/Automation-Refactoring-Plugin/tree/testing_process

11.3 Test Cases for Manual Testing of Functional Requirements

Table 11.1 – Description of test cases for functional requirements

Id	Test case	Given Data	Expected result	Actual result	Test result
1	Settings dialogue should validate entered maximum cyclomatic complexity value when apply	Given number bigger than maximum allowed value - 21	Error message shown	Method Complexity Settings Maximum allowed cyclomatic complexity for methods: 21 Methods with cyclomatic complexity over 20 are viewed as complex, potentially impacting maintainability. Please reduce maximum allowed cyclomatic complexity.	Passed
2	Settings dialogue should validate entered regex for testing naming convencion value when apply	Given not valid regular expression: `should.*_when.*_given[A-Z`	Error message shown	Test Naming Convention Regex for test method naming: should.*_when.*_given[A-Z Example: should.*_when.*_given.* Entered value is not a valid regular expression.	Passed
3	All code smell types should be detected and presented on “Analyze project” dialogue	Given project with all types of code smells	All types of code smells should be dispayed	Project Level Smells Architecture Smells Implementation Smells Test Smells	Passed
4	Settings dialogue should apply all customizations given by user	1) All inspections diables 2) Maximum cyclomatic complexity set to 20 3) RegEx for testing naming convention allows any	No code smells should be detected when running “Analyse project” window	Tools › AutoRefactor Plugin Settings Method Complexity Settings Maximum allowed cyclomatic complexity for methods: 20 Test Naming Convention Regex for test method naming: .* Example: should.*_when.*_given.* Enable/Disable Code Inspections Scattered functionality check Enum reference comparison check Objects content comparison check Object method parameter check Lengthy method check Recurring object creation check	Passed

		string: .* 4) Settings saved			
5	OOP metrics should be correctly calculated for given class	Given a class with: 1) 7 fields – 2 public, 5 private 2) 12 methods including: 4 constructors, 4 getters and 4 setters. All of them public 3) 1 child 4) Getters are used by 4 classes 5) One constructor uses method from other class 6) Contains 57 lines of code	LOC: 57 NOF: 7 NOPF: 2 NOM: 12 NOPM: 12 WMC: 8 (constructors excluded) NOC: 1 DIT: 1 (inherits the Object class) LCOM: 60 FANIN: 4 FANOUT: 1 Calculation of LCOM: Total method pairs - 66 (12*11/2), field1 – used 3 times (3 pairs), field2 – 2 times (1 pair), fieldInt1 – 2 times (1 pair), fieldInt2 – 2 times (1 pair). Total: 66 - 6=60		Passed
6	Two methods with same functionaly should be detected as “Scattered functionality” code smell	Given two methods in different classes with same functionality. But parameters have different names and one mwthod contain comments and whitespaces	Both methods highlighted as code smell “Scattered Functionality”		Passed
7	Two objects compared by reference should be detected as “Reference object comparison” code smell	Given objects s1 and s2 compared by reference: s1==s2 s1!=s2	Expressions should be highlighted as code smell “Reference object comparison”		Passed

8	Applying a quick fix for “Refence object comparison” should replace reference comparison with content comparison.	Given objects s1 and s2 compared by reference:
		
		`s1==s2`
		`s1!=s2`
			Expressions replaced by:
			`Objects.equals(s1,s2)`
			`!Objects.equals(s1,s2)`
				`public boolean method1(Object s1, Object s2) { return Objects.equals(s1, s2); }`
				`public boolean method2(Object s1, Object s2) { return !Objects.equals(s1, s2); }`
					Passed

9	Two enums compared by content should be detected as “Content enum comparison” code smell	Given enums s1 and s2 compared by content:
		
		`s1.equals(s2)`
		`Objects.equals(s1,s2)`
		`!s1.equals(s2)`
			Expressions should be highlighted as code smell “Content enum comparison”		Passed

10	Applying a quick fix for “Content enum comparison” should replace content comparison with reference comparison	Given enums s1 and s2 compared by content:
		
		`s1.equals(s2)`
		`Objects.equals(s1,s2)`
		`!s1.equals(s2)`
			Expressions replaced by:
			`s1 == s2`
			`s1 == s2`
			`s1 != s2`
				`public boolean method1(EnumComparisonEnum s1, EnumComparisonEnum s2) { return s1 == s2; }`
				`public boolean method2(EnumComparisonEnum s1, EnumComparisonEnum s2) { return s1 == s2; }`
				`public boolean method3(EnumComparisonEnum s1, EnumComparisonEnum s2) { return s1 != s2; }`
					Passed

11	Method where only one field from given parameter is accessed should be detected as “Unnecessary object parameter” code smell	Given methods where one field of parametrs acced (direct access and access by getter):
		`String method1(TestObject s1) {return s1.getField1();}`
		
		`String method3(TestObject s1) { return s1.field3;}`
			Expressions should be highlighted as code smell “Unnecessary object parameter”	`public String method1(TestObject s1) { return s1.getField1(); }`
				`public String method3(TestObject s1) { return s1.field3; }`
				Method can be refactored to accept only the property "field3" as a parameter
				`public class TestObject`
					Passed

12	Method where only one field from given parameter is accessed and method from parameter called, should not be detected as “Unnecessary object parameter” code smell	`void method2(TestObject s1) { String field1 = s1.getField1(); field1 = field1.concat(s1.getField2());}`	Expressions should not be highlighted as code smell “Unnecessary object parameter”	`public void method2(TestObject s1) {`
				`String field1 = s1.getField1();`
				`field1 = field1.concat(s1.getField2());`
				`}`
					Passed

13	Applying a quick fix for “Unnecessary object parameter” the method parameters should be refactored to accept only the property that is used	Given methods where one field of parametrs acced (direct access and access by getter):
		`String method1(TestObject s1) {return s1.getField1();}`
		
		`String method3(TestObject s1) { return s1.field3;}`
			Parametrs and usage of all parameters should be replaed by:
			`String method1(String field1) {return field1;}`
			
			`String method3(String field3) { return field3;}`
				`public String method1(String field1) { return field1; }`
				1 usage
				`public String method3(String field3) { return field3; }`
					Passed

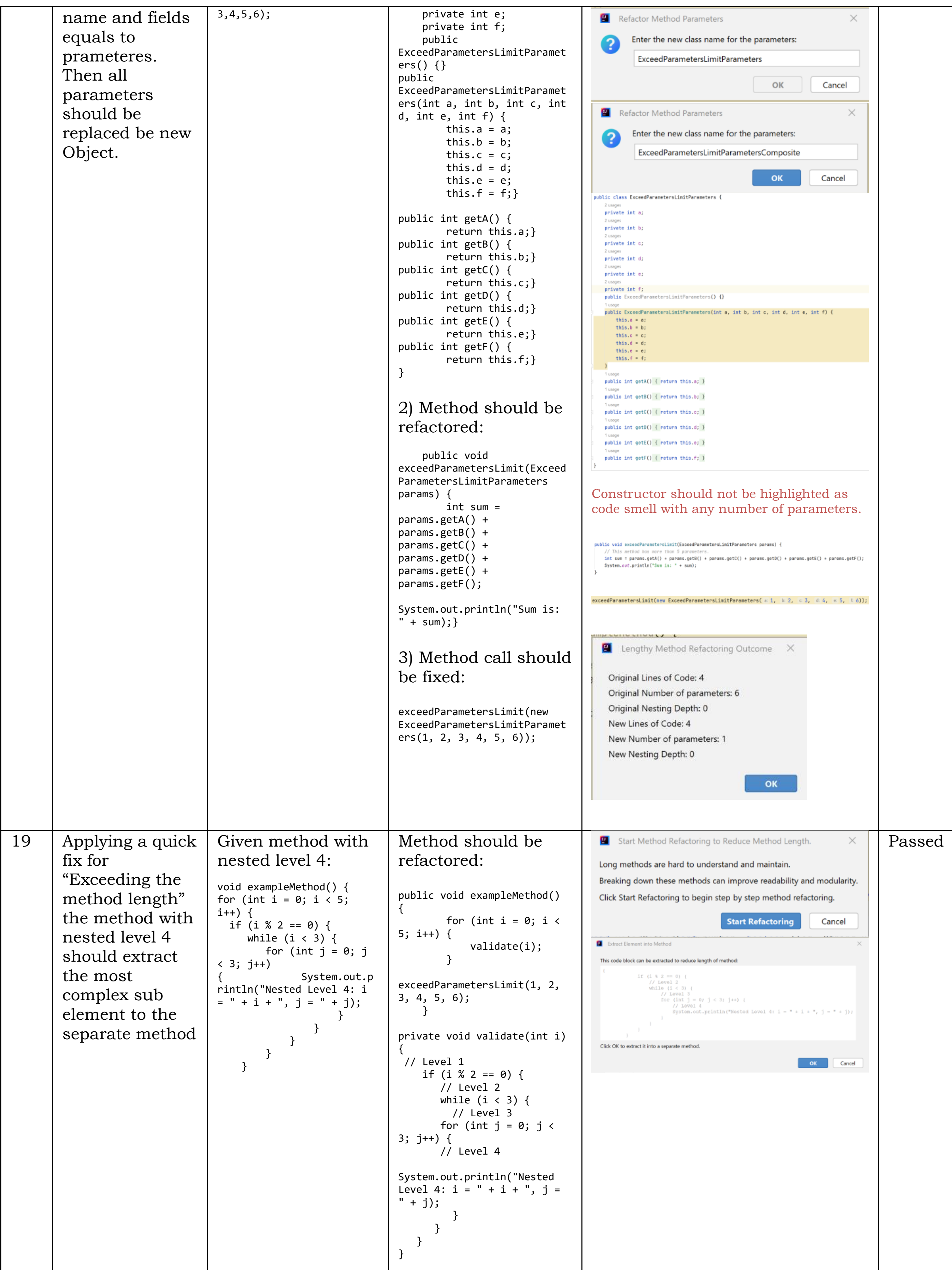
14	Method with 6 parameters should be detected as “Exceeding the method length” code smell	Given method with 6 parameters:
		`void exceedParametersLimit(int a, int b, int c, int d, int e, int f) {`
		`int sum = a + b + c + d + e + f;`
		`System.out.println("Sum is: " + sum);`
		`}`
			Method should be highlighted as code smell “Exceeding the method length”		Passed

15	Method with netsed level 4 should be detected as “Exceeding the method length” code smell	Given method with nested level 4:
		`void exampleMethod() {`
		`for (int i = 0; i < 5; i++) {`
		`  if (i % 2 == 0) {`
		`     while (i < 3) {`
		`        for (int j = 0; j < 3; j++)`
		`{            System.out.println("Nested Level 4: i = " + i + ", j = " + j);`
		`            }`
		`         }`
		`      }`
		`   }`
		`}`
			Method should be highlighted as code smell “Exceeding the method length”		Passed

16	Method, which contains more than 100 lines should be detected as “Exceeding the method length” code smell	Given method with 100 lines:
		`void methodExceedingLocLimit(boolean flag) {`
		`if (flag) {`
		`  System.out.println("Line 1");`
		`…`
		`System.out.println("Line 10");`
		`  }`
		`System.out.println("Line 11");`
		`…`
		`System.out.println("Line 100");`
		`}`
			Method should be highlighted as code smell “Exceeding the method length”		Passed

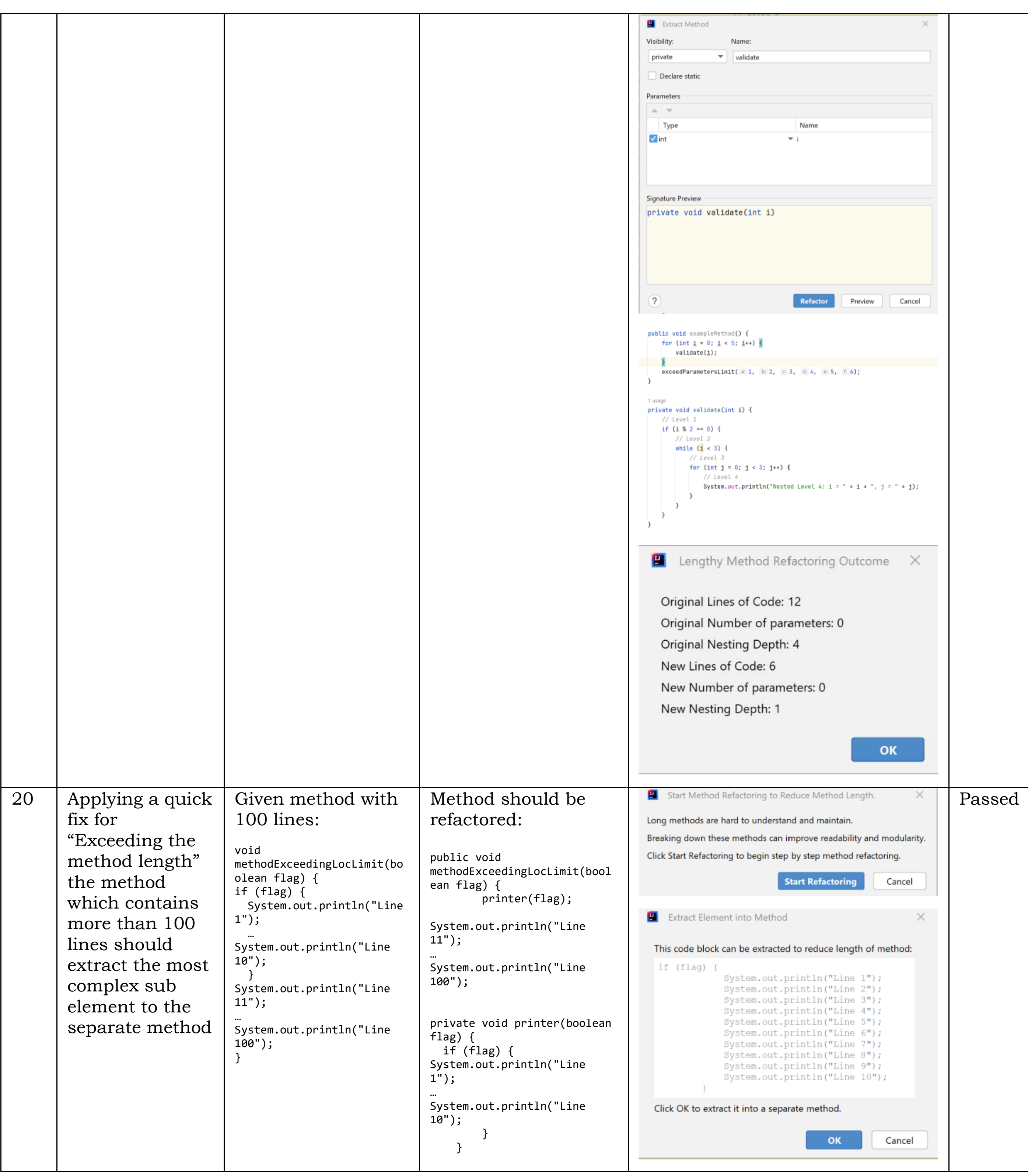
17	Method, with nested level 3 and 5 parameters should not be detected as “Exceeding the method length” code smell	Given method with nested level 3 and 5 parameters:
		`void validMethod(int number, boolean flag, String username, String hiMessage, String byeMessage) {`
		`System.out.println(hiMessage+username);`
		`if (number > 10) {`
		`System.out.println("Number is greater than 10.");`
		`  for (int j = 0; j < 10; j++) {`
		`    if (flag) {`
		`System.out.println(j);}`
		`  }`
		`} else {`
		`System.out.println("Number is 10 or less.");}`
		`System.out.println(byeMessage);`
		`}`
			Method should not be highlighted as code smell “Exceeding the method length”		Passed

18	Applying a quick fix for “Exceeding the method length” the method with 6 parameters should be created an object with, given by user,	Given method with 6 parameters:
		`void exceedParametersLimit(int a, int b, int c, int d, int e, int f) {`
		`int sum = a + b + c + d + e + f;`
		`System.out.println("Sum is: " + sum);`
		`}`
		`exceedParametersLimit(1,2,`
			1) Should be created an object (with two constructors and getters):
			`public class ExceedParametersLimitParameters {`
			`    private int a;`
			`    private int b;`
			`    private int c;`
			`    private int d;`
					Failed



	name and fields equals to prameteres. Then all parameters should be replaced be new Object.	3,4,5,6);	private int e;
			private int f;
			public
			ExceedParametersLimitParamet
			ers() {}
			public
			ExceedParametersLimitParamet
			ers(int a, int b, int c, int
			d, int e, int f) {
			this.a = a;
			this.b = b;
			this.c = c;
			this.d = d;
			this.e = e;
			this.f = f;}
			
			public int getA() {
			return this.a;}
			public int getB() {
			return this.b;}
			public int getC() {
			return this.c;}
			public int getD() {
			return this.d;}
			public int getE() {
			return this.e;}
			public int getF() {
			return this.f;}
			}
			
			2) Method should be refactored:
			
			public void
			exceedParametersLimit(Exceed
			ParametersLimitParameters
			params) {
			int sum =
			params.getA() +
			params.getB() +
			params.getC() +
			params.getD() +
			params.getE() +
			params.getF();
			
			System.out.println("Sum is:
			" + sum);}
			
			3) Method call should be fixed:
			
			exceedParametersLimit(new
			ExceedParametersLimitParamet
			ers(1, 2, 3, 4, 5, 6));
				Refactor Method Parameters
				Enter the new class name for the parameters:
				ExceedParametersLimitParameters
				OK Cancel
				
				Refactor Method Parameters
				Enter the new class name for the parameters:
				ExceedParametersLimitParametersComposite
				OK Cancel
				
				Constructor should not be highlighted as code smell with any number of parameters.
				
				Lengthy Method Refactoring Outcome
				Original Lines of Code: 4
				Original Number of parameters: 6
				Original Nesting Depth: 0
				New Lines of Code: 4
				New Number of parameters: 1
				New Nesting Depth: 0
				OK	

19	Applying a quick fix for “Exceeding the method length” the method with nested level 4 should extract the most complex sub element to the separate method	Given method with nested level 4:
		
		void exampleMethod() {
		for (int i = 0; i < 5;
		i++) {
		if (i % 2 == 0) {
		while (i < 3) {
		for (int j = 0; j
		< 3; j++)
		{ System.out.p
		rintln("Nested Level 4: i
		= " + i + ", j = " + j);
		}
		}
		}
		}
		}
			Method should be refactored:
			
			public void exampleMethod()
			{
			for (int i = 0; i <
			5; i++) {
			validate(i);
			}
			
			exceedParametersLimit(1, 2,
			3, 4, 5, 6);
			}
			
			private void validate(int i)
			{
			// Level 1
			if (i % 2 == 0) {
			// Level 2
			while (i < 3) {
			// Level 3
			for (int j = 0; j <
			3; j++) {
			// Level 4
			
			System.out.println("Nested
			Level 4: i = " + i + ", j =
			" + j);
			}
			}
			}
			}
				Start Method Refactoring to Reduce Method Length.
				Long methods are hard to understand and maintain.
				Breaking down these methods can improve readability and modularity.
				Click Start Refactoring to begin step by step method refactoring.
				Start Refactoring Cancel
					Passed



20	Applying a quick fix for “Exceeding the method length” the method which contains more than 100 lines should extract the most complex sub element to the separate method	Given method with 100 lines:
		`void methodExceedingLocLimit(boolean flag) {`
		`if (flag) {`
		`System.out.println("Line 1");`
		…
		`System.out.println("Line 10");`
		`}`
		`System.out.println("Line 11");`
		…
		`System.out.println("Line 100");`
		`}`
			Method should be refactored:
			`public void methodExceedingLocLimit(boolean flag) {`
			`printer(flag);`
			`System.out.println("Line 11");`
			…
			`System.out.println("Line 100");`
			`private void printer(boolean flag) {`
			`if (flag) {`
			`System.out.println("Line 1");`
			…
			`System.out.println("Line 10");`
			`}`
			`}`
					Passed

				`public void methodExceedingLocLimit(boolean flag) {`
				`printer(flag);`
				`System.out.println("Line 11");`
				`System.out.println("Line 12");`
				`System.out.println("Line 13");`
				`System.out.println("Line 14");`
				`System.out.println("Line 15");`
				`System.out.println("Line 16");`
				`System.out.println("Line 17");`
				
				`private void printer(boolean flag) {`
				`if (flag) {`
				`System.out.println("Line 1");`
				`System.out.println("Line 2");`
				`System.out.println("Line 3");`
				`System.out.println("Line 4");`
				`System.out.println("Line 5");`
				`System.out.println("Line 6");`
				`System.out.println("Line 7");`
				`System.out.println("Line 8");`
				`System.out.println("Line 9");`
				`System.out.println("Line 10");`
				`}`
				`}`
				
				Lengthy Method Refactoring Outcome
				Original Lines of Code: 104
				Original Number of parameters: 1
				Original Nesting Depth: 1
				New Lines of Code: 93
				New Number of parameters: 1
				New Nesting Depth: 0
				OK	

21	Two declared variables with the same content should be detected as “Repeated object creation” code smell	Given number of object creations expressions:
		
		`TestObject testObject = new TestObject("a");`
		`TestObject testObject1 = new TestObject("a");`
			Expressions should be highlighted as code smell “Repeated object creation”	`TestObject testObject = new TestObject( field1: "a");`
				`TestObject testObject1 = new TestO`
				Object is created multiple times with the same parameters
				Extract object creation to a constant Alt+Shift+Enter More actions... Alt+Enter
				`main.classes.TestObject`
				`@Contract(pure = true)`
				`public TestObject(`
				`String field1`
				`)`
				TestProject
					Passed

22	Two declared variables with the same content should not be detected as “Repeated object creation” code smell if they are created by default constructors	Given number of object creations expressions:
		
		`TestObject objectEmpty = new TestObject();`
		`TestObject objectEmpty1 = new TestObject();`
			Expressions should not be highlighted as code smell “Repeated object creation”	`TestObject objectEmpty = new TestObject();`
				`TestObject objectEmpty1 = new TestObject();`
				Object is created multiple times with the same parameters
				Extract object creation to a constant Alt+Shift+Enter More actions... Alt+Enter
				`main.classes.TestObject`
				`@Contract(pure = true)`
				`public TestObject()`
				TestProject
				
				Object creation by default constructor should not be highlighted as code smells.
					Failed

23	Two declared variables with the same content should not be detected as “Repeated object creation” code smell if one of them is modified or have an object call on it	Given number of object creations expressions:
		
		`TestObject testObject = new TestObject("a");`
		`TestObject testObject1 = new TestObject("a");`
		
		`testObject1.setField3(“b”)`
			Expressions should not be highlighted as code smell “Repeated object creation”	`TestObject testObject = new TestObject( field1: "a");`
				`TestObject testObject1 = new TestObject( field1: "a");`
				`testObject1.setField3("b");`
					Passed

<table>
<tr><td>24</td><td>Applying a quick fix for “Repeated object creation” if such constant already exists, should replace object creation by constant</td><td>Given an object creation expression:

<code>private static final TestObject CONST = new TestObject("a");</code>

<code>TestObject testObject = new TestObject("a");</code></td><td>Object creation should be refactored:

<code>private static final TestObject CONST = new TestObject("a");</code>

<code>TestObject testObject = CONST;</code></td><td><code>private static final TestObject CONST = new TestObject(field1: "a");</code>
1 usage
<code>public static void increment(Integer var) {</code>
<code>   var--;</code>
<code>   TestObject testObject = CONST;</code></td><td>Passed</td></tr>
<tr><td>25</td><td>Applying a quick fix for “Repeated object creation” if such constant not exists, should create new constant with default name and replace object creation by constant</td><td>Given number of object creations expressions:

<code>TestObject testObject = new TestObject("a");</code>
<code>TestObject testObject1 = new TestObject("a");</code>
<code>String a = "a";</code>
<code>TestObject objectA = new TestObject(a);</code>

<code>int two = 2;</code>
<code>TestObject obj1 = new TestObject(new TestSubObject(2));</code>
<code>TestObject obj2 = new TestObject(new TestSubObject(2));</code></td><td>Object creations should be refactored:

<code>private static final TestObject TESTOBJECT_CONSTANT1 = new TestObject(new TestSubObject(2));</code>
<code>private static final TestObject TESTOBJECT_CONSTANT = new TestObject("a");</code>

<code>TestObject testObject = TESTOBJECT_CONSTANT;</code>
<code>TestObject testObject1 = TESTOBJECT_CONSTANT;</code>
<code>String a = "a";</code>
<code>TestObject objectA = TESTOBJECT_CONSTANT;</code>

<code>int two = 2;</code>
<code>TestObject obj1 = TESTOBJECT_CONSTANT1;</code>
<code>TestObject obj2 = TESTOBJECT_CONSTANT1;</code></td><td><code>private static final TestObject TESTOBJECT_CONSTANT1 = new TestObject(new TestSubObject(field2: 2));</code>
3 usages
<code>private static final TestObject TESTOBJECT_CONSTANT = new TestObject(field1: "a");</code>
1 usage
<code>public static void increment(Integer var) {</code>
<code>   var--;</code>
<code>   TestObject testObject = TESTOBJECT_CONSTANT;</code>
<code>   TestObject testObject1 = TESTOBJECT_CONSTANT;</code>
<code>   String a = "a";</code>
<code>   TestObject objectA = TESTOBJECT_CONSTANT;</code>

<code>   int two = 2;</code>
<code>   TestObject obj1 = TESTOBJECT_CONSTANT1;</code>
<code>   TestObject obj2 = TESTOBJECT_CONSTANT1;</code></td><td>Passed</td></tr>
<tr><td>26</td><td>Method with cyclomatic complexity 11 (maximum allowed value set to 10) should be detected as “Exceeding cyclomatic complexity” code smell</td><td>Given method with cyclomatic complexity 11:

<code>public String cyclomaticComplexity11(int score) {</code>
<code>   if (score < 0 || score > 100) {</code>
<code>   throw new IllegalArgumentException("Score must be between 0 and 100.");</code>
<code>   }</code>

<code>String grade;</code>
<code> if (score >= 90) {</code>
<code>   grade = "A";</code>
<code> } else if (score >= 80){</code>
<code>   grade = "B";</code>
<code> } else if (score >= 70){</code>
<code>   grade = "C";</code>
<code> } else if (score >= 60){</code>
<code>   grade = "D";</code>
<code> } else {</code>
<code>   grade = "F";</code>
<code> }</code>

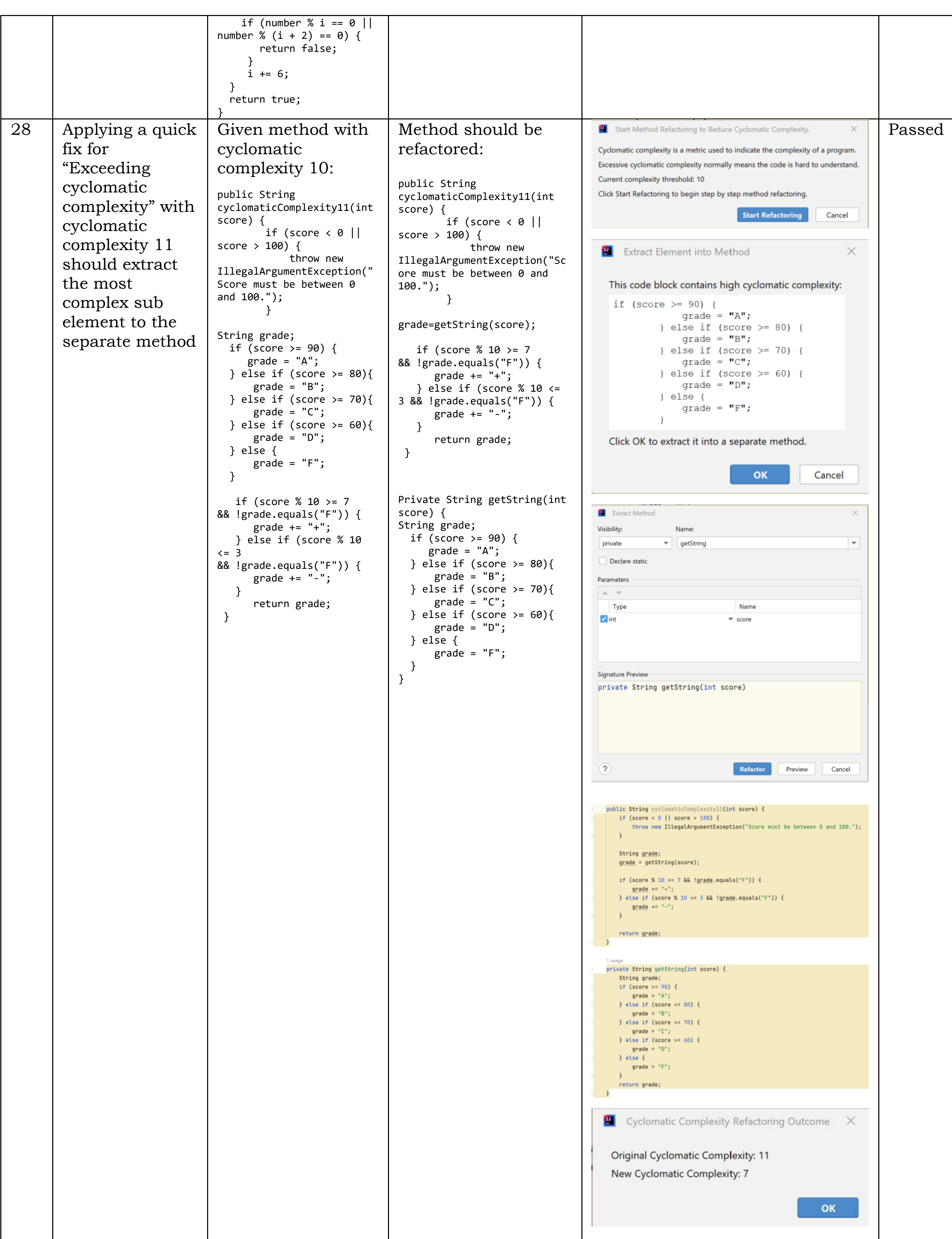
<code>   if (score % 10 >= 7 && !grade.equals("F")) {</code>
<code>   grade += "+";</code>
<code>   } else if (score % 10 <= 3 && !grade.equals("F")) {</code>
<code>   grade += "-";</code>
<code>   }</code>
<code>   return grade;</code>
<code>}</code></td><td>Method should be highlighted as code smell “Exceeding cyclomatic complexity”</td><td><code>public String cyclomaticComplexity11(int score) {</code>
<code>   if (score < 0 || score > 100) {</code>
<code>   throw new IllegalArgumentException("Score must be between 0 and 100.");</code>
<code>   }</code>

<code>   String grade;</code>
<code>   if (score >= 90) {</code>
<code>   grade = "A";</code>
<code>   } else if (score >= 80) {</code>
<code>   grade = "B";</code>
<code>   } else if (score >= 70) {</code>
<code>   grade = "C";</code>
<code>   } else if (score >= 60) {</code>
<code>   grade = "D";</code>
<code>   } else {</code>
<code>   grade = "F";</code>
<code>   }</code>

<code>   if (score % 10 >= 7 && !grade.equals("F")) {</code>
<code>   grade += "+";</code>
<code>   } else if (score % 10 <= 3 && !grade.equals("F")) {</code>
<code>   grade += "-";</code>
<code>   }</code>

<code>   return grade;</code>
<code>}</code>
Method exceeds recommended length or complexity. Nesting depth exceeds 3, current nesting depth: 4.
Cyclomatic complexity exceeds the allowed limit: 11. Refactor the method to reduce complexity for better code maintainability.
Refactor method to reduce excessive cyclomatic complexity Alt+Shift+Enter More actions... Alt+Enter</td><td>Passed</td></tr>
<tr><td>27</td><td>Method with cyclomatic complexity 10 (maximum allowed value set to 10) should not be detected as “Exceeding cyclomatic complexity” code smell</td><td>Given method with cyclomatic complexity 10:

<code>public boolean cyclomaticComplexity10(int number) {</code>
<code> if (number <= 1) {</code>
<code>   return false;</code>
<code> }</code>
<code> if (number <= 3) {</code>
<code>   return true;</code>
<code> }</code>
<code> int i = 5;</code>
<code> while (i * i <= number) {</code></td><td>Method should not be highlighted as code smell “Exceeding cyclomatic complexity”</td><td><code>public boolean cyclomaticComplexity10(int number) {</code>
<code>   if (number <= 1) {</code>
<code>   return false;</code>
<code>   }</code>
<code>   if (number <= 3) {</code>
<code>   return true;</code>
<code>   }</code>
<code>   int i = 5;</code>
<code>   while (i * i <= number) {</code>
<code>   if (number % i == 0 || number % (i + 2) == 0) {</code>
<code>   return false;</code>
<code>   }</code>
<code>   i += 6;</code>
<code>   }</code>
<code>   return true;</code>
<code>}</code></td><td>Passed</td></tr>
</table>



<table>
<tr><td></td><td></td><td>   if (number % i == 0 ||
number % (i + 2) == 0) {
       return false;
    }
    i += 6;
  }
  return true;
}</td><td></td><td></td><td></td></tr>
<tr><td>28</td><td>Applying a quick fix for “Exceeding cyclomatic complexity” with cyclomatic complexity 11 should extract the most complex sub element to the separate method</td><td>Given method with cyclomatic complexity 10:

public String
cyclomaticComplexity11(int
score) {
        if (score < 0 ||
score > 100) {
            throw new
IllegalArgumentException("
Score must be between 0
and 100.");
        }

String grade;
  if (score >= 90) {
    grade = "A";
  } else if (score >= 80){
    grade = "B";
  } else if (score >= 70){
    grade = "C";
  } else if (score >= 60){
    grade = "D";
  } else {
    grade = "F";
  }

   if (score % 10 >= 7
&& !grade.equals("F")) {
    grade += "+";
   } else if (score % 10
<= 3
&& !grade.equals("F")) {
    grade += "-";
   }

    return grade;
 }</td><td>Method should be refactored:

public String
cyclomaticComplexity11(int
score) {
        if (score < 0 ||
score > 100) {
            throw new
IllegalArgumentException("Sc
ore must be between 0 and
100.");
        }

grade=getString(score);

   if (score % 10 >= 7
&& !grade.equals("F")) {
    grade += "+";
   } else if (score % 10 <=
3 && !grade.equals("F")) {
    grade += "-";
   }
    return grade;
 }

Private String getString(int
score) {
String grade;
  if (score >= 90) {
    grade = "A";
  } else if (score >= 80){
    grade = "B";
  } else if (score >= 70){
    grade = "C";
  } else if (score >= 60){
    grade = "D";
  } else {
    grade = "F";
  }
}</td><td></td><td>Passed</td></tr>
</table>

				Note: methods are highlighted because of other code inspection	

29	Test method with not matching naming should be detected as “Test method naming convention not followed” code smell	Given a test method name:
		`@Test`
		`public void testMethod4`
		
		Given test method convention regex:
		`should[A-Z].*_when[A-Z].*_given[A-Z].*`
			Method signature should be highlighted as code smell “Test method naming convention not followed”		Passed

30	Test method with matching naming should not be detected as “Test method naming convention not followed” code smell	Given a test method name:
		`@Test`
		`public void shouldReturnConcatenatedString_whenMethod2_givenTestObject`
		
		Given test method convention regex:
		`should[A-Z].*_when[A-Z].*_given[A-Z].*`
			Method signature should not be highlighted as code smell “Test method naming convention not followed”	`@Test`
				`public void shouldReturnConcatenatedString_whenMethod2_givenTestObject() {`
				`//given`
					Passed

31	Not test method with not matching naming should not be detected as “Test method naming convention not followed” code smell	Given a test method name:
		`public void validMethod`
		
		Given test method convention regex:
		`should[A-Z].*_when[A-Z].*_given[A-Z].*`
			Method signature should not be highlighted as code smell “Test method naming convention not followed”	`public void validMethod(int number, boolean flag, String username, String hiMessage, String byeMessage) {`
				`System.out.println(hiMessage+username);`
					Passed

32	Applying a quick fix for “Test method naming convention not followed” should rename method acourding to user input	Given a test method name:
		`@Test`
		`public void testMethod4() {`
		
		Given test method convention regex:
		`should[A-Z].*_when[A-Z].*_given[A-Z].*`
		
		Given new name:
		`@Test`
		`public void shouldConcatenateStarings_whenMethod4_givenTwoStrings`
			Method name should be replaced:
			
			`@Test`
			`public void shouldConcatenateStarings_whenMethod4_givenTwoStrings() {`
				`@Test`
				`public void testMethod4() {`
				Test names should follow the "should[A-Z].*_when[A-Z].*_given[A-Z].*" format to describe the tested method.
				Rename test according to naming convention
				More actions...
				`main.classes.ObjectMethodParameterTestUnitTest`
				`public void testMethod4()`
				TestProject
				
				Rename Test Method
				Enter the new name for the method:
				should[A-Z].*_when[A-Z].*_given[A-Z].*
				OK Cancel
				
				Rename Test Method
				Enter the new name for the method:
				houldConcatenateStarings_whenMethod4_givenTwoStrings
				OK Cancel
				
				`@Test`
				`public void shouldConcatenateStarings_whenMethod4_givenTwoStrings() {`
				`//given`
					Passed

11.4 Test Cases for Manual Testing of Nonfunctional Requirements

Table 11.2 – Description of test cases for nonfunctional requirements

Id	Requirement	Actual result	Test result
1	The plugin must respond to user interactions within 5 seconds for codebases of up to 10,000 lines of code	Automated performance testing was conducted using scripts that simulated user interactions with the plugin across various scenarios within IntelliJ IDEA. The biggest response time was 3,2 seconds when tested with codebases of 11562 lines of code. This indicates excellent responsiveness and confirms that the plugin can handle typical user operations efficiently.	Passed
2	The plugin must operate within a memory limit of 1GB of RAM	To test the consumed RAM, IntelliJ IDEA's built-in profiler was utilized, revealing that the plugin used 355 MB of heap memory and 433 MB of non-heap memory, totaling 788 MB of RAM usage.	Passed
3	The plugin should interact seamlessly with other IntelliJ IDEA plugins and extensions, ensuring that combined functionality does not introduce conflicts or performance issues	Integration testing was carried out in a development environment loaded with multiple popular IntelliJ IDEA plugins to check for any compatibility issues or conflicts. Specifically, the environment included the following plugins: “Git Integration”, “Lombok Plugin”, “Maven Helper”, “CodeGlance”, “VisualVM Launcher” These plugins were chosen because they cover a range of functionalities, thus providing a robust testing ground for ensuring that the new plugin operates seamlessly within a complex plugin ecosystem. The plugin interacted seamlessly with other plugins, indicating strong compatibility and no introduction of performance issues or conflicts.	Passed
4	The plugin should be compatible with IntelliJ IDEA 2021.1 and later (current latest version 2024.1)	The plugin was tested on the most versions of IntelliJ IDEA, ranging from the oldest supported version (2021.1) to the latest (2024.1). Tested versions include: 2021.1, 2021.2, 2022.1, 2022.2, 2023.1, 2024.1. Compatibility with all versions was confirmed, demonstrating that the plugin functions correctly across multiple versions of the IDE without any issues.	Passed
5	The plugin shall support Java 8 and above, accommodatin	Functional tests were run on environments configured with Java versions starting from Java 8 to the latest (21), to ensure that all plugin functionalities were supported.	Passed

	g developers using modern Java language features	The plugin was compatible with mentioned before Java versions, all features operated as expected, validating its ability to accommodate developers using modern Java features.	
6	Plugin only collects essential information required for its functionalities	The plugin collects only data related to its settings, without saving or transmitting code snippets or involving any third-party services. This ensures full compliance with data minimization and privacy best practices, maintaining the integrity and security of the user's data.	Passed
7	All data processing and storage occur within the local development environment	No external data transmissions or storage were used; all data processing and storage occurred locally, ensuring compliance with security and privacy standards.	Passed
8	The plugin installation package's target size cannot be more than 10MB	The installation/distribution package measured 7088KB, effectively under the 10MB threshold. This confirms that the plugin is compact and facilitates quick and efficient downloads and installations.	Passed

11.5 Public User Documentation

https://butonsusumom.github.io/Automation-Refactoring-Plugin/

11.6 Plugin Distribution Link

https://plugins.jetbrains.com/plugin/24498-autorefactor/versions/stable