International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Development of Optimization based Refactoring Sequencing Approach in Object-Oriented System

Ritika Maini[1],Dr. Amandeep Kaur[2]

Guru Granth Sahib World University Fatehgarh Sahib

ABSTRACT:

In the Software Development Life Cycle (SDLC), software maintenance holds a crucial position. However, maintaining software has become increasingly challenging due to evolving requirements. Programmers continuously modify the source code to adapt to these changes and enhance software features. Consequently, a significant portion, approximately eighty percent, of project costs is allocated to maintenance activities. Identifying code smells plays a vital role in reducing maintenance costs [1]. Code smells are structural defects within the software, indicative of poor software design, making it difficult to maintain. They point out design issues in software applications. The presence of code smells hampers code comprehension and potentially leads to increased changes and susceptibility to faults. Detecting and removing code smells from the source code enhances the developer's understanding of the code.

Keywords: Software Development Life Cycle, Code smells, Refactoring, Software quality.

Introduction :

Software Maintenance

The objective of software maintenance is to update along with alter application programmes afterwards they are delivered with a view to fix bugs as well as to boost the action[1].The programme needs to be changed whenever feasible. If a software product has been given to the customer, it can be changed through a process called software maintenance. The software maintenance becomes important to:

1. Rectify errors.
2. Improve every situation connecting to other systems
3. Programs must be able to accommodate many types of hardware, software,system features, and telecommunications equipment.
4. Upgrade outdated software.

Software Maintenance Process steps:

1. The first and foremost step in maintenance is to plan, which involves software preparation, problem diagnosis, and product configuration management.

2. The second step problem analysis process which entails to determine the validity of the problem, examining it, developing a solution, and lastly obtaining all necessarysupport in order to submit an application for modification.

3. The third step is to enquire the Updation that the user wants to include in the software application that made the quest before accepting the process.

4. [1] https://teamairship.com/4-types-of-software-maintenance-to-know/
5. The next process relocation, which is used when software needs to be moved fromone platform to another without losing functionality [2].

Thus, hence the updated as well as changed product is handed to the customer afterwards. The updated software needs to be changed whenever feasible. The figure1 shows the software maintenance step by step.

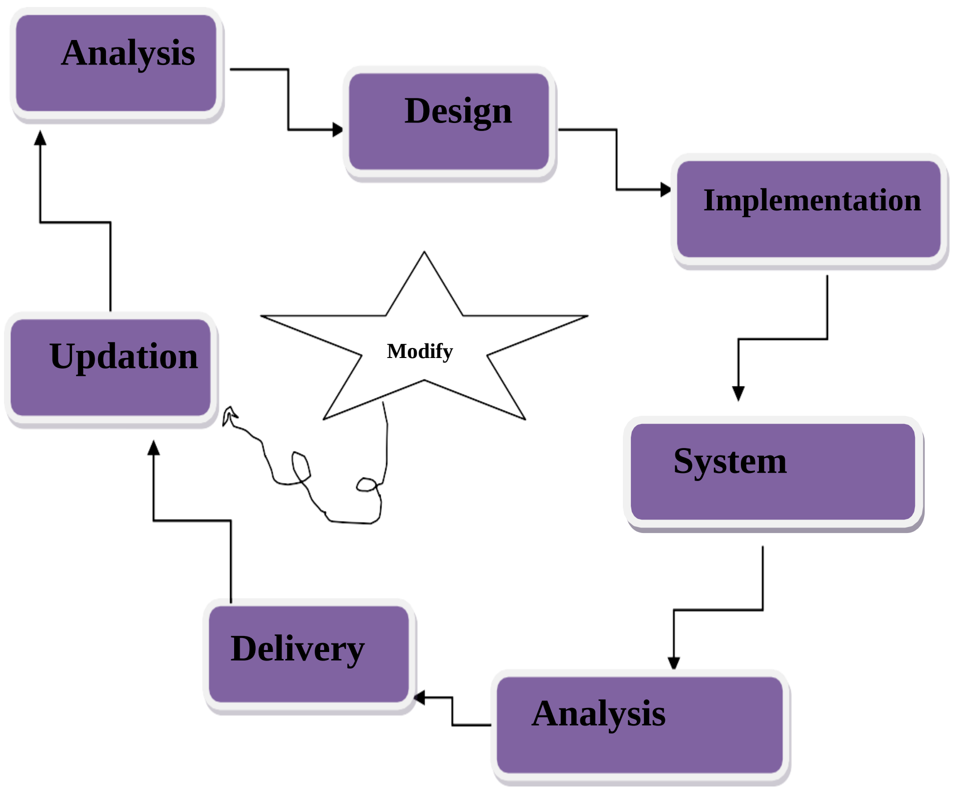

Figure 1: Software Maintenance Process

Classification of Software Maintenance:

Every type of software maintenance has its own action to perform. A particular part of application might have to go through once, twice or every classification of maintenance during its lifetime.

1. **Adaptive**: To ensure the system's adaptability with changing environments,some adjustments are made to it.
2. **Perfective:** Checks for fine-tuning of all system features, functionality, andcapabilities in order to optimize system performance.
3. **Corrective:** Detection and rectification of faults as well as inaccuracy in thepresent solution to ensure the system's proper operation.
4. **Preventive:** Aids in the prevention of potential system flaws.

These are four categories of software that help ensure the software as a whole operates as intended is as shown in figure 2 with the amount of percentage of their respective purpose.

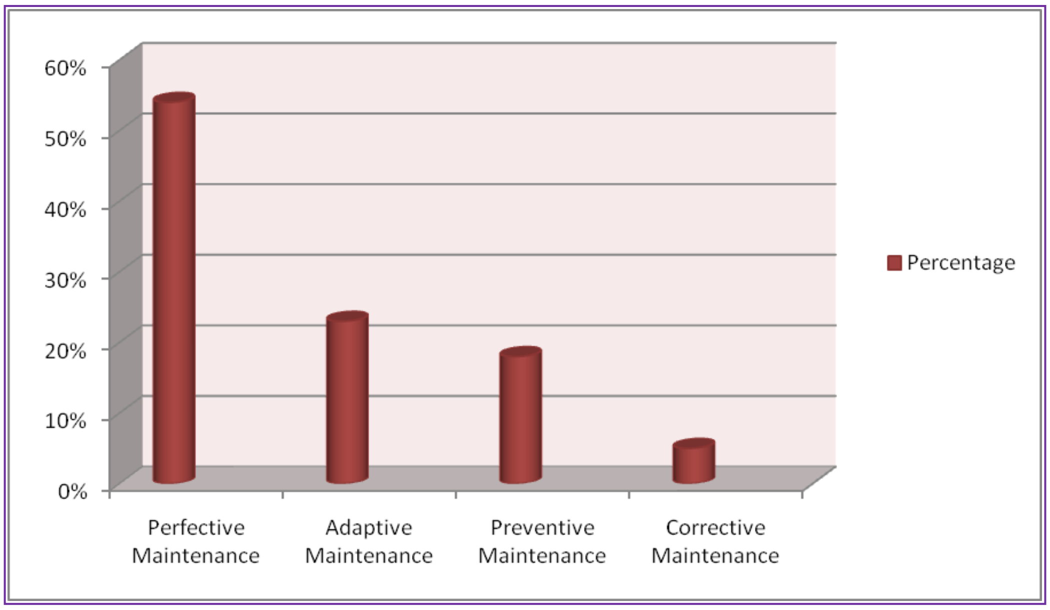

Figure 2: Software Maintenance Categories

Software Quality

"Software quality measures how well the software conforms to the design and how well the software is designed ". It is frequently referred to as software's suitability. The software lifecycle includes software maintenance. You must decide what it is and why it is vital in your software business because it is the key stage of the application development process. Therefore, software development lifecycle does not begin and stop with the development of the software. Your software will need to be monitored and maintained even after it has been deployed [3]. The makers must ensure that the software functions smoothly as long as it is functioning and consumers are using its features. This is where software maintenance comes into play. This procedure guarantees that the software continues to function properly and prepares the system for future changes. It also improves the software's features and makes any necessary changes to improve the product.

The software maintenance is the last phase of life cycle of software. The main objective is to change and timely modifications in the delivered software application so as to monitor bugs, repair faults and maintain its utmost production. Software maintenance is an extensive word which refers to development, fixing the bugs, the shifting of old process to new one, and the improvement of existing functionality. So with the demand of changing world the software needs updations and requires criteria for budgeting, capturing and scheduling timely developments. Thus, the main software reliability depends on this period of Application development life-cycle (ADLC) as 50-75 % of the accuracy depends on this software maintenance and the cost of bearing this. As a result, concentrating on maintenance can help you save money [4].

Implications for the Decline in Software Quality

Deteriorating software quality refers to the software's inability to carry out tasks as intended by the user. These criteria may or may not be functional. There are various causes of low-quality software, the low-quality software can be varied, and some of the prominent factors are:

1. **Insufficient domain knowledge:** The unfortunate truth that the majority of developers lack expertise in the field in which software is created may be the biggest factor in low software quality. They will eventually get greater knowledge of the domain, but much of it will come from fixing errors brought on by incorrect interpretations of the functional requirements.

2. **Non sensual timetables:** Sound software development practices are frequently sacrificed by developers in order to meet unrealistic deadlines, and the outcomes are rarely favorable. Developers who are under pressure to work quickly face greater challenges or have less time to collect them.

3. **Poorly crafted software:** Most software development activities entail modifying or improving existing code to an extent of at least two thirds. Studies have indicated that trying to understand what is happening in the code takes up around half of the time spent updating already-existing software. Code that is overly complex is frequently impossible to understand and changing it typically results in a great deal of errors to unintended bad side effects.
4. **Low grade procurement procedures:** The majority of substantial inter programs are built and run by workgroups, most or all of which may be contracted to other businesses. As a result, the acquiring company frequently lacks knowledge of and/or control over the software's quality.

5. **Lack of technological expertise:** The majority of developers are skilled in a variety of programming languages and technologies. Modern multi-tier corporate systems, however, are a complex maze of numerous programming languages and various software platforms. These tiers comprise a desktop application, application logic, and data processing. Few developers are familiar with all of these programming languages and technological platforms, and their false assumptions about how those other platforms operate are a major cause of the non-functional flaws that lead to catastrophic failures, data breaches, and unauthorized access while in use [5].

Need of Software Maintenance

Software maintenance is required for a variety of reasons, as noted below:

1. **Errors to be removed:** The most crucial aspect of service is the rectification of errors, sometimes known as "bugs." It is critical that the software runs without errors. Putting this activity as a priority is recommended. This procedure entails looking for and correcting flaws in the code. Hardware, operating systems and any piece of software can all have issues. This will be done in order to preserve the procured old software's functions.

2. **Expanding opportunities in a changing world:** Such product must be analyzed in order to improve present functions as well as must establish a method that is adaptive in updated programmes. It improves programme capabilities, work patterns, hardware updates, compilers, and all other components of the system's workflow. Improve the performance of your system through some updated functions as well as utilizing maintained software.

3. **Remove obsolete functions:** Features which can't be helpful and occupy space unnecessarily. Thus, it is required so as that remove obsolete functions. Existing coalition as well as unnecessary coding is deleted, hence further functions are created using the updated equipments and machinery. As a result of this adjustment, the system is more adaptable to changing situations.
4. **4.Enhancement of performance:** So, to fulfill further requirements, the attainment of system should be upgraded. Software maintenance includes data and encoding constraints, as well as re-engineering. As a result, hence impacts are not vulnerable. It's a feature that evolves to prevent dangerous behaviors like harm, mutilate etc [6].

Thus, maintenance in software must not an option; it is required. Take, for example, an automobile. If you don't offer it, it can bring number of issues throughout the year. The expense of poor car maintenance will be significantly higher. Similarly, ignoring system maintenance will limit your ability to grow your business to its full potential. Maintainability of increasingly massive and complex software systems is a fundamental concern in today's information society. Many methods have been proposed to forecast software maintainability, but actual quantification of the relationship between maintainability and measurable software properties such as code smells has eluded researchers. The identification of software defects has become a standard way for identifying user interface development flaws that could create difficulties during expansion and improvement. As a result, the general consensus is that code having smells must be re-factored to avoid or mitigate such issues. Refactoring, on the other hand, comes with both costs and dangers. As a result, empirical proof assessing the link between code smell and there is need of maintenance of software.

Types of Software Metrics/Measuring Maintainability

"What you cannot measure, you cannot control" Thus software needs to be maintained and measured timely with the purpose to get utmost enforcement of the software. In order to calculate the size of software there needs to check its reusability, efforts, repetitive data and complexity also. For instance, despite the fact that two distinct systems have the same features, different programmers can explain the various levels of difficulty. A programmer's level of experience will increase the application developed by the developer is probably smaller in comparison to greater in terms of functionality and effort, but with increasing complexity, redundancy, and reusability. Procedural approach is further extended to form Object-oriented approaches. Object oriented approach deals with real world entity that's why we must focus on it whereas Java supports procedural as well as object- o r i e n t e d approach. The below table 1shows the various classifications of software metrics:

1. Traditional approach
2. Object oriented approach

1. **Traditional approach:** This approach is applied to make projects that require imperative programming. They use simple procedure in this approach such as interpretation, composition of application and experimentation. This approach is based on project size and types of projects to be worked. The development of larger projects sometimes needs a larger duration and large capital. The most commonly used traditional metrics include Line of Code (LOC), Cyclomatic Complexity (CC) and some other shown in table 1. The main problem in this approach is the classic life cycle on which the project is based.

2. **Object oriented approach:** The projects are object-oriented programming based as it deals with real world entities. It applies Unified Modeling Language (UML) code such as case, class diagram, sequence diagram and many like this. This approach is more time consuming as compared to traditional one and also the more costly is. The most common object-oriented metrics are Metrics for Object Oriented Design (MOOD metrics), Quality Model for Object Oriented Designmetrics (QMOOD) and others as shown in table 1. The three traditional design activities are there of analysis, design and implementation [6].

Metrics	
Traditional	Object -Oriented
Depth of conditional nesting	Number of children
Cyclomatic complexity	Coupling between objects
Fan-in/Fan-out	Weighed methods per class
Fog index	Number of overriding operations
Lines of code	Response for a class

Table 1: Software Metrics Hierarchy

The following are the two other sorts of Software Metrics:

1. Internal metrics: Internal metrics seem to be measurements that are utilized to measure properties some of which are apparently more important to a software developer includes the performance of software, planning of different works done during developments in software and during productivity and many more jobs to be done. For example, Lines of Code (LOC) measure.

2. External metrics: External metrics is considered when more important external jobs to be done like to increase the performance of your software. These types of metrics are important when some reporting is done on an application or cluster. For example, cloud monitoring etc.

Code Smell

Code smell is different from bugs or errors. Other than that, code smells are evident offensive of application development integrals which decrease the quality of the code. In the mid-1980s, programming like C, C++ slowly undergo into the scene of technical advancement. So the encrypting is seen like just a task of producing results, regardless of the encryption or method used [7]. Coding is about improve the application entirely as it just execute, stays for more time, and can be seen. It does not give any hint that the application will run or not ; it can still give the result; it can take more time to process the code and increase the chances of default and mistakes while developing he code. Code smells signal a more serious problem, although they are detectable or detectable quickly, as the term implies [8]. The finest smell is that it becomes easy to locate but leads to an intriguing difficulty, such as classes with data but no behavior. With the use of tools, codes smells can be simply obtained. Every characters with in code base that points to a more serious issue is called a code smell. Code smells are contravention to the principles of application development which lower standard the software rather than being faults or errors.

Code smells may cause processing to take longer, increase the risk of failure and mistakes, and create the programme extra susceptible in the direction of problems in the future, but they don't provide the full information specifically about the working of the software. The figure 3 shows the life cycle of code smell.

Taxonomy of Code Smell

The taxonomy is created in order to know more about the code smells and to get information how they are connected to each other. Code smells are divided into seven types as of Object-Oriented Abusers, Couplers, Encapsulator, Bloaters, Change Preventers, Dispensable and some other general types [9-10].

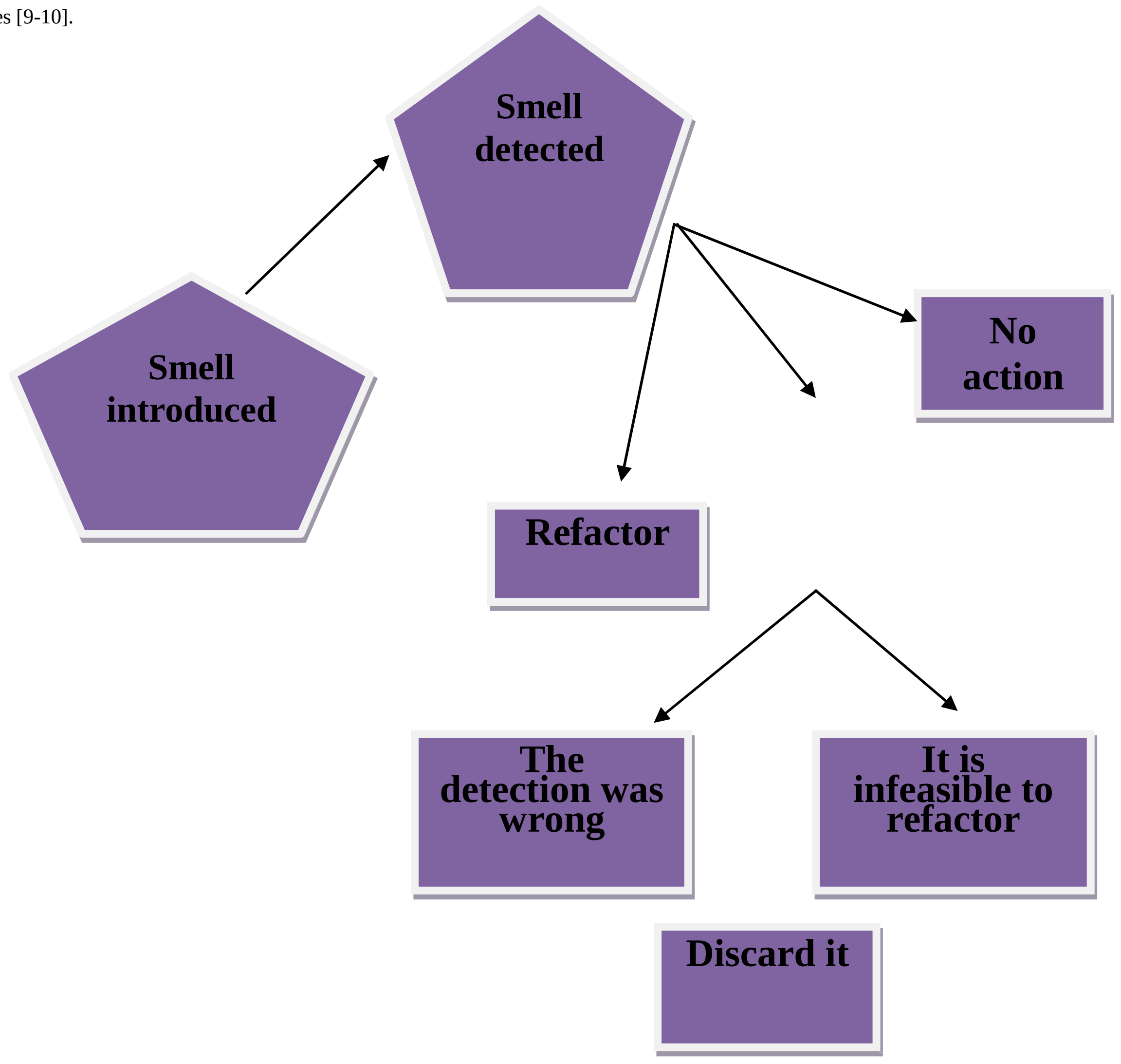

Figure 3: Code Smell Life Cycle

REFERENCES:

1. Sjøberg, D. I., Yamashita, A., Anda, B. C., Mockus, A., & Dybå, T. (2012). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, *39*(8), 1144-1156.
2. Chaparro, O., Bavota, G., Marcus, A., & Di Penta, M. (2014, September). On the impact of refactoring operations on code quality metrics. In *2014 IEEE International Conference on Software Maintenance and Evolution* (pp. 456-460). IEEE.
3. Mohan, M., & Greer, D. (2019). Using a many-objective approach to investigate automatedrefactoring. *Information and Software Technology*, *112*, 83-101.
4. Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison- WesleyProfessional.
5. Baqais, A. A. B., & Alshayeb, M. (2020). Automatic software refactoring: a systematic literaturereview. *Software Quality Journal*, *28*(2), 459-502.
6. Kaur, S., Kaur, A., & Dhiman, G. (2021). Deep analysis of quality of primary studies on assessing the impact of refactoring on software quality. *Materials Today: Proceedings*.
7. Ouni, A., Kessentini, M., Sahraoui, H., & Boukadoum, M. (2013). Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering*, *20*(1), 47-79.
8. Kaur, S., & Singh, P. (2019). How does object-oriented code refactoring influence software quality? Research landscape and challenges. *Journal of Systems and Software*,*157*, 110394.
9. Singh, S., & Kaur, S. (2018). A systematic literature review: Refactoring for disclosing code smells in object-oriented software. *Ain Shams Engineering Journal*, *9*(4), 2129-2151.
10. Mkaouer, M. W., Kessentini, M., Cinnéide, M. Ó., Hayashi, S., & Deb, K. (2017). A robust multi- objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering*, *22*(2), 894-927.