[[1]](#footnote-0)

The Research of Software Rearchitecting (April 2023)

Xiaotian Zhang,Shi Chen,Haoyu Liu,Jie Chen,Haoran Xu,Chenchao Gao,Haoxuan Sun,Yunlong Tang

*Abstract*—Software design and development are iterative processes that require constant adaptation to changing requirements and maintenance. Software rearchitecting is a key software technology that improves software comprehensibility, maintainability, and scalability. It is a process that improves the structure of software without changing its function and external visibility. The benefits of software rearchitecting are numerous, including the reduction of the overall complexity of the system or software. Complexity can arise from program logic, architecture design, system deployment, testing, and technology. Basic and composite rearchitecting methods have been defined, as well as software rearchitecting assistance tools. In our paper, we introduce several Methods and Techniques of Software rearchitecting.

Keywords: Software rearchitecting, code smells, software architecture, software development, complexity.

# INTRODUCTION

S

oftware design and development is an iterative process, facing changes in requirements, additions and deletions of functions, and adjustments to applications at any time. It is difficult to establish a software architecture that can adapt to various needs in the initial stage of software design and development. When facing new requirements and software maintenance, it mainly relies on software rearchitecting. An important principle of software rearchitecting is to only change the internal structure of the software without changing its external behavior. Software rearchitecting can significantly reduce software maintenance costs and extend the service life of the software.

# Body

## Definition of Software Rearchitecting

Software rearchitecting refers to the transformation of software in order to improve the structure of the software, improve clarity, scalability and reusability without changing the function and external visibility of the software. In short, rearchitecting is improving the design of an already written software. Which means that the original functions and behaviors of the improved software will not be changed, only the code inside the program will be changed. rearchitecting is carried out at the source code level, and the purpose is to make the original code have a good program structure after rearchitecting. Software rearchitecting is a key software technology to improve software comprehensibility, maintainability and scalability. It is a research hotspot and an important practice in the field of software engineering. Software rearchitecting not only changes the situation that the software is overly dependent on the previous design, but also strives to obtain an appropriate software structure, while maintaining the simplicity and flexibility of the design.[1]

## The role and benefits of software rearchitecting

The biggest function and benefit of software rearchitecting is to reduce the overall complexity of the system or software. The complexity in software development is of course as low as possible. In the actual software development process, in addition to the complexity of logic and design, complexity involves all aspects.

The complexity of program logic

The complexity of linear sequential execution is 1, and after the branch appears, it must be multiplied by the number of branches. Branches can be conditional judgments or loops. Therefore, avoiding the occurrence of branches as much as possible is an important means to reduce the complexity of program logic. If program branching is unavoidable, try to place program branching at the highest logical level. The purpose of this is to avoid divergent branches in the lower layer processing. Divergent branching will dramatically increase the complexity of the program. The higher the complexity, the harder the program is to maintain.

Complexity of Architecture Design

Architecture design involves module design and system design. It is necessary to extract some common modules or subsystems as much as possible, and these common functions may be called by other business modules or systems. When calling these public functions, the simpler the better, and the caller does not need to care about the specific internal implementation, only needs to know how to use it. The purpose of this is to focus on the design of the business or service code.[2]

Complexity of System Deployment

System deployment consists of several different phases such as development phase, testing phase and production phase. No matter which stage it is, the fewer steps to deploy, the less error-prone it is. Some systems naturally require the configuration of many commands. If this is the case, it is necessary to write a batch file to simplify the deployment steps of external users and turn multiple steps into one step.[3]

Complexity of Testing

Testing is divided into White Box Testing and Black Box Testing. The complexity of White Box Testing is directly related to the complexity of the code level. The higher the complexity of the code level, the higher the complexity of White Box Testing. An important issue that needs to be paid attention to in White Box Testing is not to make the code of this part of White Box Testing deviate from the design of the actual business code. The complexity of Black Box Testing comes from business requirement analysis. To have very clear documentation, it needs to write very clearly about the test steps and expected results.

Complexity of Technology

The development trend of technology is generally that the more it develops, the simpler it is, and the more powerful it is. During the design and development process, avoid using old technology. Simplify our learning process, improve development efficiency, and enhance the maintainability of the entire project.[4]

## Research Status of Software Rearchitecting

Research Progress on Software Rearchitecting

Basic Rearchitecting Methods and Composite Rearchitecting Methods Opdyke defined a set of basic rearchitecting methods for C++ language, which were improved by Roberts et al. Exploration and organization of bad program structures, such as redundant code exploration and elimination, class hierarchy organization, cross-cutting concern exploration and encapsulation, etc. Program understanding methods and tools, such as the Ovation system and the Rigi reverse engineering environment. Rearchitecting methods for typical designs, such as manual rearchitecting of design patterns in programs and automatic extraction of UML interaction diagrams from C++ code. Software rearchitecting assistance tools, such as C++ rearchitecting tools and Smalltalk Refactory Browser.[5]

Identification and Detection of Code Smells

Metric-based Code Smell Detection Firstly, the metrics representing the characteristics of the source code are measured numerically, such as attribute number, code lines, parameter number, method quantity, and class quantity, etc., and then the threshold of indicators is combined to determine whether it is code smell or not. Representative works include the inCode plugin, quality attributes of coupling and cohesion proposed by Veerappa et al., code smells related to JavaScript language proposed by Fard et al., Chen et al. use three different filter strategies to specify metric thresholds, and Jiang et al. propose a code smell detection method based on the distribution model of class length and the metric of cohesion for LargeClass.

Rule-based Code Smell Detection Combining rules and logical expressions to analyze the symptoms of code smells, convert them into detection rules, and finally select suitable thresholds for detection. Representative works include the detection rules of 20 code smell types for Android application platform proposed by Carvalho et al. and the DECOR tool proposed by Moha et al.

Additional Content

The Significance and Necessity of Software Rearchitecting Software rearchitecting can help developers improve software quality, reduce technical debt, enhance maintainability and scalability without changing program behavior. Especially in large-scale software systems, with the passage of time, changes in requirements and continuous iterative development, a large amount of redundant code and complex structures will be generated, and rearchitecting is needed to maintain the health of the software system.

The Impact and Harm of Code Smells Code smells can cause a decline in software quality, making it difficult to maintain and extend, increasing subsequent development costs and risks. At the same time, they also affect the readability and understandability of the code, increasing the difficulty of debugging and rearchitecting for developers.

Future Research Directions Future research directions can be explored from the following aspects: (1) more accurate methods for detecting code smells, including using deep learning, machine learning and other technologies to extract and judge code features. (2) research on visualization rearchitecting tools, which makes the rearchitecting process more intuitive and convenient through graphical interfaces and interactive operations. (3) research on rearchitecting for cloud computing and big data, aiming at rearchitecting and optimization of distributed and large-scale data processing systems. (4) research on automatic rearchitecting methods, which can improve rearchitecting efficiency and reduce human errors through automatic ways.

# Methods and Techniques of Software Rearchitecting

Search-Based Rearchitecting

Software rearchitecting is a challenging task in software engineering because finding the most effective sequence of rearchitectings is difficult. Developers need to perform multiple rearchitectings when maintaining software, and each rearchitecting may trigger the precondition for another rearchitecting. Therefore, building a reasonable rearchitecting sequence is essential to ensure software quality. Search-based rearchitecting (SBR) is a popular method for addressing this issue. It applies search-based software engineering (SBSE) to rearchitecting tasks, using software metrics as objectives to guide the search process and obtain the best rearchitecting sequence through search.

The solution space of rearchitecting sequences is enormous, making finding the best sequence challenging. However, search algorithms can efficiently explore the solution space to find the best sequence. To achieve this, the software system is transformed into a collection of software components or abstractly represented as the input of the algorithm. Rearchitecting suggestions, software quality metrics, and other information guide the algorithm, and all rearchitecting operations are mapped to vectors as the representation of solutions. The fitness function dynamically evaluates the quality of solutions to obtain the optimal or locally optimal rearchitecting sequence.

To improve the performance of SBR, various researchers have proposed different methods. For example, Amal et al. proposed a fitness function based on neural networks. The function uses a genetic algorithm to evaluate rearchitecting solutions and then trains the remaining iterations using an artificial neural network. Dea et al. proposed a PCA-NSGA-II multi-objective rearchitecting method that removes redundancy to avoid the curse of dimensionality. Mkaouer et al. proposed a new representation method for the rearchitecting problem, treating each quality attribute that needs improvement as an independent optimization objective. Mohan et al. proposed MultiRefactor, an automated rearchitecting tool based on multi-objective genetic algorithms, using four independent software metrics as optimization objectives for rearchitecting operations. Boukharata et al. proposed the WSIRem method based on multi-objective search optimization to help developers improve the modularity of service interfaces.[6]

UML-based Rearchitecting

Model Driven Software Engineering (MDSE) employs models at different abstraction levels for software development, maintenance, and evolution. Unified Modeling Language (UML) is widely used in software rearchitecting, focusing on model transformation to improve structure while maintaining quality characteristics.

Reimann et al. proposed a general rearchitecting method based on role models using the Eclipse Modeling Framework (EMF) and evaluated it with UML and Web Ontology Language (OWL). Steimann replaced constraint checking with constraint solving, transforming the role of constraints in rearchitecting.

Arcelli et al. targeted performance anti-patterns in the EPSILON platform and implemented detection rules and rearchitecting operations for UML models. Einarsson et al. applied model-to-model transformations, proposing a method to refactor UML models and their diagrams within UML tools, and implemented a prototype plugin based on Eclipse.

Arcelli et al. introduced the PADRE tool for detecting and rearchitecting performance anti-patterns in UML models. Lu et al. proposed a Search-Based OCL constraint Rearchitecting Approach (SBORA), using four semantic preservation rearchitecting operators and three object constraint language quality metric indicators to refactor specified constraint sets in UML meta-models.[7]

Code Smell-based Rearchitecting

Code smells pose challenges in software rearchitecting due to their concealed nature and the complexity of software systems. Researchers from various backgrounds are investigating techniques to address these challenges.

Bavota's team devised the "Methodbook" algorithm to eliminate Feature Envy and Inappropriate Intimacy code smells by assessing relationships between functions and identifying target classes for migration rearchitecting. They also used algorithms and graph theory to decompose God classes and extract related functions and attributes into new classes.

Mumtaz et al. studied code smells in five software systems and refactored those related to software security metrics, thus improving system security. Politowski et al. analyzed the impact of Blob and Spaghetti Code on code understandability, suggesting that these code smells should be refactored together.

Wang et al. introduced an automated rearchitecting technique based on software complex network models, employing rearchitecting preprocessing operations and clustering algorithms to redivide system modules according to the "high cohesion, low coupling" principle. They also proposed rearchitecting recommendations for code smells caused by cohesion and coupling.

Bu presented a God class detection method using deep neural networks to help developers identify rearchitecting opportunities quickly. Liu et al. created a code smell detection tool to prompt developers to refactor software systems, thus enhancing software quality and reducing development costs.[8]

Test-Driven Rearchitecting

Test-driven rearchitecting adjusts test code before rearchitecting source code, ensuring proper implementation of continuous small-step rearchitecting, leading to refined code and low-defect software systems.

Xuan et al. proposed B Rearchitecting, dividing test cases into smaller fragments for better dynamic analysis support. Gao et al. established a mapping relationship between rearchitecting and test case failures, guiding automated test case repair, analyzing failure causes, and assessing rearchitecting impacts on software interfaces, also proposing a method for analyzing rearchitecting impact on regression test cases.

Chu et al. emphasized test case rearchitecting in pattern-based software development, proposing a four-phase method for design pattern test case rearchitecting, initially assigning pattern structures based on roles for functional and non-functional requirements, aiding test case creation.

# Conclusion

Software design and development are iterative processes that require constant adaptation to changing requirements and maintenance. The biggest function and benefit of software rearchitecting is to reduce the overall complexity of the system or software, including The complexity of program logic, Complexity of Architecture Design, Complexity of System Deployment, Complexity of Testing, Complexity of Technology.

Software rearchitecting technology can be used in many situations. This paper mainly discusses how to solve Code Smells problem. Code Smells degrades software quality, makes it difficult to maintain and expand, and increases subsequent development costs and risks. At present, there are mainly two methods to detect Code Smells. One is Metric-based Code Smell Detection. Firstly, the values representing the characteristics of source code, such as the number of attributes, the number of lines of code, the number of parameters, the number of methods and the number of classes, are measured, and then the threshold value of indicators is combined to determine whether it is Code Smells. The second is Rule-based Code Smell Detection: combining rules and logical expressions, analyze the symptoms of Code Smells, convert them into detection rules, and finally select the appropriate threshold for detection.

In Software rearchitecting, it is difficult to find the most efficient rearchitecting sequence. Therefore, there are four main reconstruction methods in the current research: Search-Based Rearchitecting, UML-based Rearchitecting, Code Smell-based Rearchitecting and Test-Driven Rearchitecting.

Under the current development background of software rearchitecting technology, after investigation and study, we believe that the future research direction of software reconfiguration can be developed from the following aspects: more accurate methods for detecting code smells, research on visualization rearchitecting tools, research on rearchitecting for cloud computing and big data and research on automatic rearchitecting methods.

References

1. H. Andrade, C. Berger, I. Crnkovic and J. Bosch, "Principles for Re-architecting Software for Heterogeneous Platforms," 2020 27th Asia-Pacific Software Engineering Conference (APSEC), Singapore, Singapore, 2020, pp. 405-414, doi: 10.1109/APSEC51365.2020.00049.
2. H. Andrade and I. Crnkovic, "A review on software architectures for heterogeneous platforms", 2018 25th Asia-Pacific Software Engineering Conference (APSEC), pp. 209-218, Dec 2018.
3. S. R. Chowdhury, M. A. Salahuddin, N. Limam and R. Boutaba, "Re-Architecting NFV Ecosystem with Microservices: State of the Art and Research Challenges," in IEEE Network, vol. 33, no. 3, pp. 168-176, May/June 2019, doi: 10.1109/MNET.2019.1800082.
4. H. Andrade, L. E. Lwakatare, I. Crnkovic and J. Bosch, "Software challenges in heterogeneous computing: A multiple case study in industry", 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 148-155, Aug 2019.
5. Selmadji, A., Seriai, AD., Bouziane, H.L., Dony, C., Mahamane, R.O. (2018). Re-architecting OO Software into Microservices. In: Kritikos, K., Plebani, P., de Paoli, F. (eds) Service-Oriented and Cloud Computing. ESOCC 2018. Lecture Notes in Computer Science(), vol 11116. Springer, Cham. <https://doi.org/10.1007/978-3-319-99819-0_5>
6. Awsan M. Mohammed, Salih O. Duffuaa,A tabu search based algorithm for the optimal design of multi-objective multi-product supply chain networks,  
   Expert Systems with Applications,Volume 140,2020,112808,ISSN 0957-4174
7. R. J. Bril and A. Postma, "An architectural connectivity metric and its support for incremental re-architecting of large legacy systems," Proceedings 9th International Workshop on Program Comprehension. IWPC 2001, Toronto, ON, Canada, 2001, pp. 269-280, doi: 10.1109/WPC.2001.921737.
8. E. Dimitrov, A. Schmietendorf and R. Dumhe, "UML-based performance engineering possibilities and techniques," in IEEE Software, vol. 19, no. 1, pp. 74-83, Jan.-Feb. 2002, doi: 10.1109/52.976944.

1. [↑](#footnote-ref-0)