iipp publishing

Science of Maintenance

Journal homepage www.m-sci.rs

ISSN 2787-3072

1-2 (2021) 1

Maintenance and maintainability within agile software development

Milena Vujosevic Janicic

University of Belgrade, Faculty of Mathematics, Belgrade, Serbia

Abstract

In agile software development, software maintenance is present almost from the beginning of software development life cycle and is usually considered together with software evolution. Making changes in software, either as corrective, preventive, adaptive or perfective maintenance, comes with additional risks and costs. In this paper, we discuss formal static software verification approaches and their influence on triggering software maintenance processes and on lowering costs and risks through automating regression verification checks. We also discuss software maintainability as a key software quality attribute in context of the overall software quality and describe the effects of software refactoring to maintainability. We present formal static verification approaches that can support the refactoring process.

Key words

software maintenance;
maintainability;
software verification;
software refactoring;

Corresponding author

milena@matf.bg.ac.rs

Article info

Original Scientific Paper
Received 3 March 2021
Accepted 7 May 2021
Online available 15 September 2021
Open access article CC BY license

1. Introduction

Over the past years, IT industry is rapidly evolving and is one of the most growing industries worldwide. Software is developed for the large variety of different consumer devices and purposes, including internet of things, virtual and augmented reality, gaming and entertainment, smart environments, consumer healthcare, artificial intelligence and big data, and communication technologies. With an increasing software production trend, software engineering processes that emphasize an acceleration of software delivery are getting more attention, and agile software development approaches are being rapidly enhanced. Common software development life cycle (SDLC) includes planning, analysis, design, implementation, testing/integration and maintenance [49, 62]. While in traditional models of software development, all these phases used to be clearly separated over involved actors and allocated time, within agile software development, where software engineers and customers work together on the products, software maintenance becomes tightly integrated into other development processes, and software maintenance and software evolution are commonly considered together [74].

According to ISO/IEC 14764 standard [40], software maintenance is divided into four categories. Corrective and preventive categories are concerned with fixing existing bugs in software. If a bug is reported by customers, then fixing it corresponds to corrective maintenance, while if a bug is observed and fixed by a software developer or maintainer, this corresponds to preventive maintenance. Adaptive and perfective categories are concerned with proactive software enhancements. Adaptive maintenance keeps the software usable, by making modifications ac- cording to evolving changes in the overall software environment. Perfective maintenance keeps the software quality, especially quality related features that influence software maintainability. Maintainability is a software quality attribute that represents the capability to efficiently in- corporate the code changes [75] corresponding to software maintenance categories, that include correcting faults, improving performance issues, adapting the software to a changed customer requirement or a changed environment. Maintainability assumes several subattributes, including testability, as the process of correcting software issues may introduce new ones. Therefore, software testing is a part of software maintenance, while high testability implies high maintain- ability and vice versa, i.e. these two software quality attributes mutually reinforce each other. As software design and the overall quality decreases over time [52, 65], it is necessary to apply different techniques to preserve maintainability. Software refactoring is a process of improving design of an existing code and is an important part of keeping maintainability through software evolution [29].

In this paper, we discuss software evolution and maintenance within agile SDLC with an emphasis on continuous delivery (Section 2). We present connections between maintenance and verification and validation processes and show influences of the newest formal verification approaches on software maintenance (Section 3). We give a brief overview of software quality attributes and characterize maintainability as one of the key software quality attributes that unifies modularity, reusability, analyzability, modifiability, and testability (Section 4). We present software refactoring principles as a driving force for keeping maintainability through software evolution, giving an insight to novel tools and research that can support the refactoring process (Section 5).

2. Software maintenance and evolution within SDLC

Software maintenance and evolution are strongly connected concepts [74, 61] and are usually considered together. While maintenance is a common engineering concept, software evolution is recognized in 1965, and is used to describe the way the software grows and evolves over time [38]. The main goals of maintenance are to fix and prevent different kinds of failures. Maintenance used to be considered as a set of activities that are conducted after delivery of software to customers, but with modern software development approaches that includes early and continuous software delivery, maintenance can also be present during software development. The goals of software evolution are to evolve and enhance software by implementing new functionalities or by adapting and improving the existing functionalities. In general, maintenance does not introduce major changes to the system, while evolution can introduce substantial changes.

Software development life cycle (SDLC) defines processes that are followed in software pro- duction [49]. The main aim is to produce and maintain high-quality software that corresponds to customer expectations, within time and cost estimates. International standard ISO/IEC 12207 defines software life-cycle processes including both initial development and maintenance of software. Each project has a unique combination of requirements, environment, involved engineers and customers. SDLC model should always be carefully chosen and adapted to a concrete project. Basic SDLC models include the waterfall model, the V-model, the itera- tive/spiral model and prototyping [62]. Modern SDLC models emphasize continuous software delivery. The waterfall model and the iterative model are presented in Figure 1.

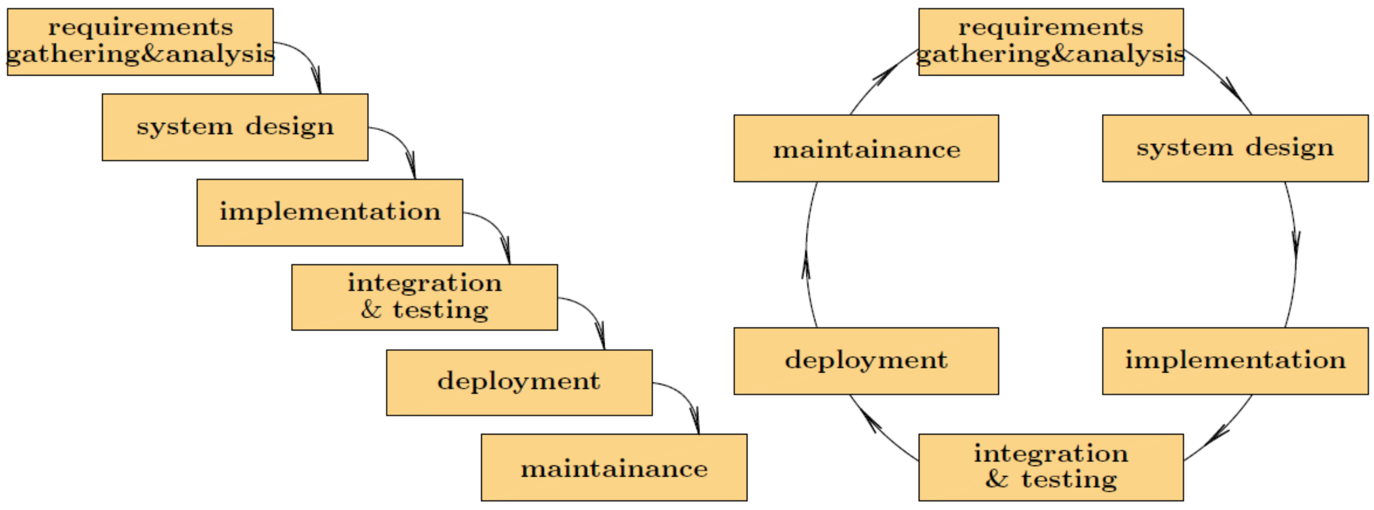

Figure 1: The waterfall SDLC model (left) and iterative SDLC model (right)

The waterfall model is the first developed SDLC model [62]. It defines basic software development phases that are still present in all modern structured SDLC models. In the waterfall model, the phases are accomplished sequentially, there is no overlapping between the phases and the outcome of one phase is the input for the next phase. SDLC starts with requirement gathering and analysis, which specifies the system that should be developed. It continues with system design which defines the overall system architecture, built within the implementation phase and integrated and tested within the next phase. Finally, once the system is integrated and tested, it is deployed in the expected environment, and after its deployment, the maintenance phase begins. This model predicts possible issues that should be fixed and maintenance is done to deliver all the necessary changes to the customer. Waterfall model is applicable only if a set of very strict conditions are satisfied, and in such cases has many advantages, but also important disadvantages including a big risk and uncertainty. The V-model [62] is an extension of the waterfall model which emphasize the importance of verification and validation by adding a testing phase for each mentioned development phase. While testing after each phase reduces the overall risks of a project failure, this model is also considered to be with high risk. Software prototyping model emphasize the importance of software validation in a timely manner, and relies on building software prototypes which display the user oriented functionalities of the built system. The built prototype might be completely unconnected to the implementation of the final product. The customer reviews are then used to build the final software solution.

In traditional iterative model, different development phases are repeated sequentially, while in the spiral model, the waterfall phases are repeated iteratively, building the overall system in an incremental manner, until the software is finally completed and released, and the maintenance phase begins. However, within modern iterative models, maintenance and evolution activities usually exist within software development and are closely intertwined with development.

Traditional view of SDLC where maintenance is a single step at the end of the development cycle is misleading [64]. Modern development usually incorporates maintenance within standard development activities and development requirements usually imply rapid and continuous software delivery. Software maintenance can also have its own life cycle, SMLC, that roughly consists of understanding the code, modifying the code and revalidating the code [7]. Different variations of SMLC are available in literature [1, 14, 15, 87].

There are two maintenance standards, ISO/IEC 14764 [40] which is a part of the standard ISO/IEC 12207 [41], and the maintenance standard IEEE/EIA 1219 [20]. These standards organize the maintenance activities within software development phases, and maintenance is a part of problem identification, analysis, design, implementation, testing and delivery.

Methodologies that emphasize rapid and continuous software delivery include rapid appli- cation development model and agile model. Rapid application development model is based on prototyping and iterative development but with no strict and specific phases within the development process. Agile SDLC is an iterative and incremental model which focuses on process adaptability, customer satisfaction and rapid delivery. The software product is usually broken into small parts which can be incrementally built within approximately two weeks. Within this period of time, all software development phases are conducted, the current version of software is built and usually deployed and also given to customers. Customers can review and use each deployed version of software and therefore the maintenance phase is an integral part of software development process. In this scenario, it can be very difficult to split between development and maintenance, as development is guided by customer's experience with the current version of software. Common maintenance issues, such as corrections and enhancements of software, are a crucial part of software development and define software evolution.

An important difference between maintenance and development activities that allows their differentiation is that development is usually driven by well defined system requirements while maintenance is driven by ongoing events [48]. Events that can trigger software maintenance include, for example, a change of request from a customer, software failure or usage related is- sues. Fixing a bug can be initiated both by a customer and by a software developer, depending on a nature and consequences of the discovered issue. Discovering a bug in a system or realizing needs for changes can occur at any time, and, therefore, events that trigger maintenance activities cannot be predicted in advance.

3. Enhancing maintenance with formal verification approaches

Fixing existing bugs and preventing new bugs are activities corresponding to corrective part of software maintenance. On the other hand, finding and fixing bugs are activities that correspond to software verification and validation (V&V) processes. Therefore, software maintenance and V&V are strongly connected.

Testing corresponds to dynamic program analysis and is usually used as a synonym for V&V. There are different approaches to testing [58, 22]. Only a subset of existing testing techniques is applied on a software project, depending on the system requirements and the overall project characteristics. Testing can be done on the lowest implementation level, corresponding to unit testing, on integration level, corresponding to component and integration testing, and on system level, corresponding to system testing. System testing also includes exploratory testing, acceptance testing and different kinds of nonfunctional testing techniques, like configuration testing, capacity testing, compatibility testing, performance testing, regression testing, security testing, and installation testing.

There are many tools and frameworks that assist and automate different kinds of testing. For example, support for automated running of unit tests is usually part of integrated software development environment, while for automating testing of web applications there are tools like Selenium [21], Katalon [43] and TestComplete [66]. On the other hand, there are aspects of software that cannot be automatically assessed, like, for example, learnability [77].

However, testing is only one part of V&V approaches, and there are also important approaches for checking correctness of software without its execution, namely by using static program analysis. Static program analysis includes code reviews and automated approaches. Code reviews are very important for achieving high software quality, especially in the context of code maintainability [19]. By code reviews it is checked if there are some errors in code logic, if all important cases are covered by implementation, if the code is covered with appropriate test cases, if the code follows corresponding project's coding standards, if there exists a better solution or more efficient algorithm that can be used. Code reviews can be more or less formal, and include formal inspections, over-the-shoulder reviews, e-mail pass-around, tool-assisted reviews and pair programming. There are many tools that assist code review process, like Phabricator [39], Gerrit [33], and Review board [6].

Automated approaches for static program analysis include code linters and more sophisticated tools, i.e. formal static analysis tools, usually based on traditional artificial intelligence approaches. Code linters look for stylistic errors, suspicious constructs, security issues, code smells and usually can spot only some simple programming errors. Linters are usually based on syntax analysis [54], while more sophisticated tools perform semantic analysis of code.

There are different formal approaches for automated checking of semantic properties of a given program. Properties of interest include, for example, finding bugs that can raise run-time errors, like buffer overflows, division by zero or type mismatches. Most common approaches are abstract interpretation [24], symbolic execution [47], and model checking [18] and there are many tools based on these approaches.

Abstract interpretation scales well on huge code repositories. It does not give precise results, i.e. it can have false positive results but cannot give witnesses for violated properties. Abstract interpretation-based tools, in the absence of reported bugs, guarantee the absence of possible bugs in the examined code. Therefore, the usage of such tools is required in development of safety critical software. Examples of tools include Astree [10], Coverty [8] and Polyspace Bug Finder [26].

Symbolic execution generalizes testing and corresponds to static execution of a program with symbolic instead of concrete values [3]. It is used for both automated bug finding and automated test-case generation. Symbolic execution uses SAT/SMT solving [9] or custom built solvers. Examples of tools based on symbolic execution are KLEE [13], Microsoft's PEX [73] and SAGE [34].

Model checking is a formal verification approach, originally developed for checking correct- ness properties of hardware systems, but it is now widely used for software systems as well [18]. Model checking can be explicit-state or symbolic. Explicit-state model checking enumerates and explores all possible states of a system, while symbolic model checking represents sets of states symbolically and uses SAT/SMT solving or binary decision diagrams. Model checking can give witnesses for violated properties that can be used for automated test case generation. Tools based on model checking include CBMC [17], LLBMC [53], ESBMC [23], Java Path-Finder [76].

Combination of formal approaches can also be used for automated checking of semantic properties. For example, LAV [79, 80, 83, 82, 78, 67] is a publicly available, open source, general purpose LLVM-based [51] tool. For constructing correctness conditions, it combines different techniques including symbolic execution, model checking and SAT encoding of program's control-flow. As an underlying reasoning machinery, for solving conditions, it uses SMT solvers and supports usage of Z3 [25], Yices [27], MatSat [12] and Boolector [11].

The presented approaches and tools can help in finding bugs and trigger and support both corrective activities of software maintenance. However, fixing one bug may introduce another bug and software maintenance should prevent such situations, i.e.

should preserve functional software equivalence between the old and the fixed version of code. Checking if some functional property is preserved between two versions of code is called regression verification [70, 69, 2] and is usually done by regression testing. Formal verification of equivalence of two programs is an undecidable problem. However, some simple but still very useful properties can be automatically proved and formal static analysis tools can be used to contribute to this issue. For example, one such property is k-equivalence and the tool LAV is successfully used in the context of regression verification [82, 67].

4. Maintainability as a key quality attribute

Software quality is a degree to which software product possesses the desired set of software quality attributes [5, 86]. Software quality is achieved through:

Assurance — incorporating quality aspects in everyday work, and

Control — ensuring that the obtained outputs are of the desired quality.

Software quality assurance [50, 36, 31] subsumes processes that have in focus acquiring and keeping software quality. It monitors and assures that all other processes, methods and activities used within a project ensure desired quality of software. The desired quality may be defined by software requirements, or can be defined as an externally quality standard like, for example, ISO 9000 or ISO 15504. Software quality control includes software verification and validation processes. Depending on the purpose and aims of the software, each software quality attribute may have different importance level. Software quality attributes, defined by standard ISO 25010 are presented in Fig. 2.

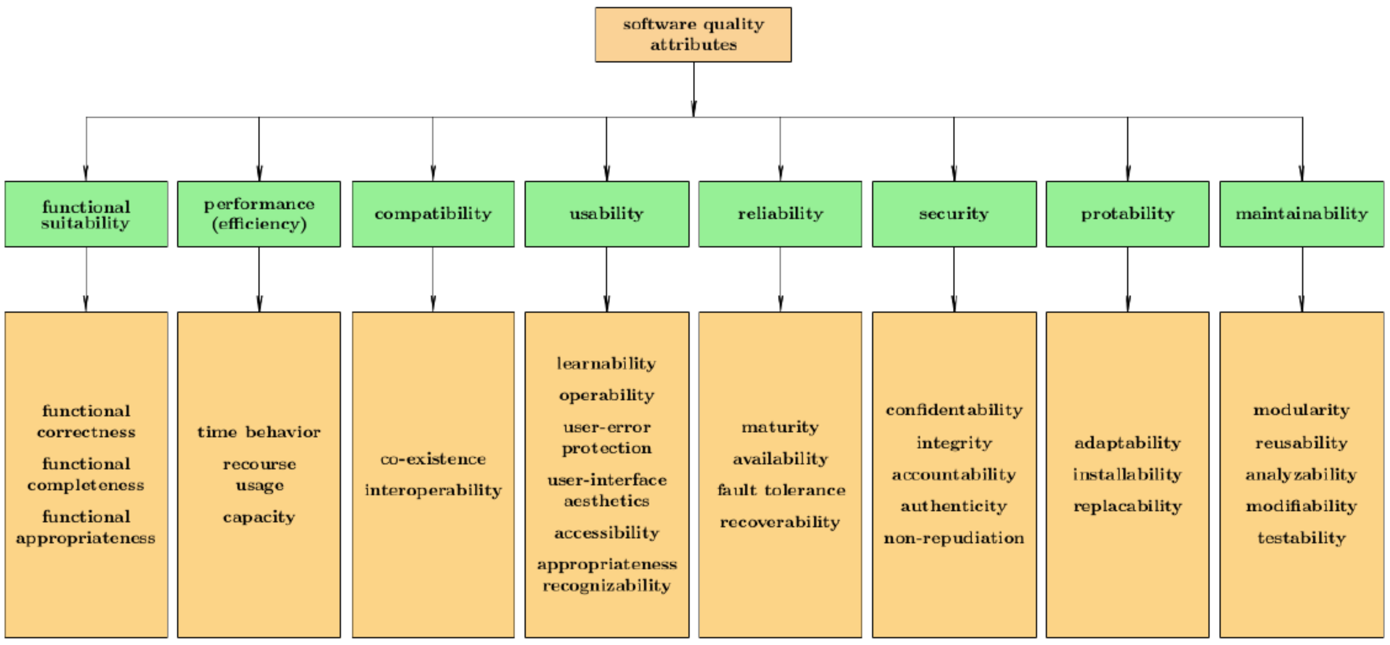

Figure 2: ISO/IEC 25010 categorization of software quality requirements [42]

Maintainability is considered as a key quality attribute [75] as it describes the capability of software to be modified and improved, i.e. its possibility to outlive unpredictable future challenges. Making a change in software requires:

1. understanding the software;
2. finding locations in software that need to be changed;
3. making desired changes;
4. checking that changes have not broken the existing code.

Maintainability addresses the easiness of all these steps. As maintainability is a static quality attribute, it cannot be assessed by testing, and different static metrics have to be considered. Some examples of static software metrics that are important in the context of maintainability are coupling (quantitative measure of interdependencies between different modules), cohesion (quantitative measure of interconnection between functions or objects of a same module), cyclomatic complexity (quantitative measure of the number of linearly independent control flow paths) and size (number of lines of code). Namely, low coupling, high cohesion, low cyclomatic complexity and small size are characteristics of a maintainable software.

According to ISO 25010 standard, maintainability is divided into five subattributes: modularity, reusability, analyzability, modifiability and testability.

4.1. Modularity

Modularity refers to a degree in which logical partitioning into independent and interchange- able modules is present within software. Breaking software into modules (units, components) allows hiding the overall software complexity (by abstraction and interface). Modules should be of relatively small size, with low cyclomatic complexity, high cohesion and there should be low coupling between modules. Such modules then can be separated and flexibly recombined in a numerous way. Modularity assumes standardized interfaces between modules, which are emphasized within modern software architectures such as microservices [59]. Modularity im- pacts the easiness of understanding software and also the easiness of finding and implementing changes within software. It is usually set as one of the main goals of software design phase.

4.2. Reusability

Reusability refers to a degree in which components of one system can be used within other systems. There are different levels of reusability, including specification reuse, design reuse, code reuse, data reuse, application system reuse, and test reuse. Reusability is of crucial importance in the context of making desired changes [32, 56], as instead of developing new functionalities from scratch, it should always be considered if some existing components can be reused. Reusability is directly connected to modularity, as high quality modularity is a prerequisite to software reusability. Reusability is also coupled with analyzability, including interface complexity and documentation, as it is important to easily understand the component that should be reused. Main benefits of software reusability include an increase of productivity, costs minimization, quality improvement, development acceleration and process risk reduction [30, 63, 4].

4.3. Analyzability

Analyzability refers to easiness of analyzing and understanding the software. Therefore, it impacts the first two steps of making a change in software (understanding the software and finding locations in software that need to be changed). Analyzability is connected to modularity, as good modularity reduces complexity and therefore improves analyzability. It is also connected to reusability, as reusing existing software components can make the analysis of code much easier. High cohesion and low coupling positively influence code analyzability, as in such code programming logic concerning one aspect of system is strongly localized. Similarly, low size and cyclomatic complexity also positively influence code analyzability, as it is easier to analyze and understand smaller and non-complex portions of code. For high analyzability, the code should be well documented, and should adopt and follow chosen coding standards. Following coding standards should be enforced by code reviews and by code linters. High analyzability is a consequence of both high quality design and how quality coding.

4.4. Modifiability

Modifiability refers to the easiness of implementing desired changes within software, without introducing new bugs and issues. Coupling is a key metric for modifiability, as high coupling implies changes that are spread out the code and that easily introduce new bugs. There are different aspects of coupling that should be considered, including return value coupling, parameter coupling, and shared variable coupling [44]. System modularity improves modifiability, as modularity hides system complexity and with low coupling makes changes more localized. For increasing modifiability, there are different techniques that reduce intracomponent coupling, like introducing layers that separate different technical responsibilities, for example, separating into different layers responsibilities such as business logic and data access. Layers give the opportunity of separating maintenance issues and also positively influence reusability. However, layers do not positively influence performance, as the many interfaces and communication between components slow down the efficiency. Modifiability is very tightly connected with testability, as without good testability it is not easy to check if a modification implied some new issues.

4.5. Testability

Testability refers to the easiness of checking if changes have not broken the existing code. Testability is influenced by project characteristics. Gathering test cases is usually done manually, but in some special cases tests can be generated automatically [13, 34, 72, 71, 55]. However, if an oracle function is not available, for example when the result of computation is not known in advance, then testing such application is more difficult and only some special kinds of testing, like metamorphic testing [16], are available. Software can be tested on different levels, and some levels can be automated, like unit testing, while some kinds of testing have to be done manually, like acceptance or exploratory testing. If large portions of testing can be run automatically, that improves testability. Testability directly influences the end users as high testability im- pacts deliverability. Software that can be thoroughly tested in a shorter amount of time can get to users faster and without unexpected failures. Also, developers benefit from getting feedback more often, and that allows timely fixes and fast iterations. Test driven development emphasize importance of testability. Testability can be measured by different metrics, for example, by the number of available test cases, by the time needed for all tests to be run, and by different test coverage criteria.

4.6. Enhancing maintainability

To achieve high maintainability, it is important to include it as a goal of each phase of software development life cycle. Maintainability can be enhanced by adopting modern coding standards, documentation standards and tools that support automated test case generation and running. However, as software evolves, it gets more complex, and the maintainability may decrease if additional care is not taken in order to keep and improve the maintainability over the time.

An important technique that can be used for keeping and improving maintainability is software refactoring.

5. Improving maintainability with software refactoring

Software refactoring corresponds to changes of the code structure that preserve functional equivalence and aim to make software easier to comprehend and to modify [52, 29]. Software refactoring is a term usually used in object oriented programming, while software restructuring is used in imperative programming. Although object oriented and imperative programming differ, there are some important refactoring/restructuring techniques that are very similar and used in both cases. In the following text, we will use the term refactoring.

Refactoring improves software quality concerning all quality subattributes related to soft- ware maintainability [45]. The catalogue of software refactorings includes more than sixty different refactoring techniques [29]. These techniques can be divided according to different problems with code structure, usually called code smells, such as:

Huge functions/methods or modules/classes that should be separated. Separating code in such context decreases complexity and therefore directly improves modularity, reusability and analyzability. It can also positively influence testability, as smaller portions of code are easier to test, and indirectly increase modifiability.

Incomplete or incorrect application of programming principles, including object-oriented principles, complex switch statements or sequences of if statements, wrong usage of code hierarchy or its absence, and alternative classes/modules with different inter- faces. Refactorings used for improving these code features positively influence reusability, analyzability and modifiability.

Existence of ripple effect that manifests with necessity of making multiple different changes within a single class/module or necessity of making a single change to multiple classes/- modules. Refactorings used for improving cohesion in this context positively influence analyzability and modifiability, and can indirectly influence reusability and modularity.

Code redundancy, like duplicated code, comments, dead code, and speculative generality. Refactorings used for removing code redundancy improve analyzability, modifiability and testability.

Coupling between classes/modules, such as usage of the internal fields and methods of an- other class/module and intensive usage of message chains. Refactorings used for removing coupling improve reusability, analyzability and modifiability, and indirectly modularity and testability.

Software refactoring is an everyday practice within agile software development [67]. Within software refactoring, programmers should systematically make small changes in code in order to preserve software equivalence [52, 29, 57]. Software refactoring techniques usually affect small and localized portions of code while some refactorings are used just for preparing code to the application of some other refactorings. Each refactoring step should be followed by thorough testing such that if a bug is introduced during the refactoring process, it is noticed and fixed immediately. Good code coverage by tests is essential for the refactoring process. However, different surveys showed that refactoring may involve additional costs and risks [46, 84], and that programmers need tools that automate and support this process [65, 85, 37].

Simple code refactorings are integral parts of integrated software development environments, for example variable renaming or function renaming. However, for each such change, a program- mer should manually check and verify that it is done correctly. Checking functional equivalence between two versions of code is an undecidable problem, but different approaches are developed to assist in this process [28, 35, 82, 81, 60]. Formal software verification techniques can be used to enhance refactoring process. For example, the tool LAV is successfully used for supporting refactoring steps that include simultaneous changes of code that includes different programming languages, namely C/C++ and embedded SQL [67, 68, 83]. For automating support of such refactorings it is necessary to precisely model both imperative programming within C/C++ programming language and declarative programming present with SQL code.

6. Conclusions

With an increasing popularity of agile software development, guided by rapid and continuous software delivery, software maintenance activities become an integral part of everyday software development processes. Therefore, improving quality of software maintenance results and effects is becoming even more important.

Maintenance includes fixing the existing bugs and preventing the new ones and is therefore strongly connected with software verification activities. It can be both triggered and supported by software verification and research results concerning automating and making verification results more reliable positively influence software maintenance. For example, risks and costs that are involved with making changes within software can be reduced by using modern formal software verification approaches that support automated bug finding and automated equivalence checking.

Maintainability is a key quality attribute that should be set as a goal of each phase of software development life cycle. In addition, keeping high maintainability trough a long lasting software evolution should be supported by continuous software refactoring. Software refactoring catalogue includes a set of good practices that guide changes of code that do not modify the external software behaviour but that positively influence the internal software quality. Applying refactoring techniques involve possibility for introducing new bugs and support for regression verification is highly important to make this process reliable. Regression testing is commonly used within refactoring process, but new formal static regression verification techniques, based on automated checking of code equivalence, introduce new possibilities and increase the overall reliability of refactoring changes.

To further support reliability of maintenance outcomes, it is important to strength static formal software verification approaches and to introduce the newest research results into every- day practice. As most of the problems that are encountered in this context are undecidable, there will always be a room for additional heuristics, improvements and upgrades.

7. References

[1] Lowell Jay Arthur. Software evolution: the software maintenance challenge. Wiley-Interscience, 1988.

[2] J. Backes, S. Person, N. Rungta, and O. Tkachuk. Regression verification using impact summaries. In International Conference on Signal Processing and Integrated Networks (SPIN), pages 99–116, 2013.

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono Delia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. ACM Computing Surveys, 51(3), 2018.

[4] Rajiv D Banker and Robert J Kauffman. Reuse and productivity in integrated computer-aided software engineering: An empirical study. MIS quarterly, pages 375–401, 1991.

[5] Mario Barbacci, Mark H Klein, Thomas A Longstaff, and Charles B Weinstock. Quality Attributes. Technical report, Carnegie-Mellon University, Software engeineering Institute, Pittsburgh PA, 1995.

[6] Inc. Beanbag. Review board, 2021. https://www.reviewboard.org/, retrieved March 15nd, 2021.

[7] Keith H Bennett and Vaclav T Rajlich. Software maintenance and evolution: a roadmap. In Conference on the Future of Software Engineering, pages 73–87, 2000.

[8] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. Mc- Peak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. Communications of the ACM, 53(2):66–75, 2010.

[9] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. Handbook of Satisfiability, volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press, 2009.

[10] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Min e, D. Monniaux, and X. Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real- Time Embedded Software, invited chapter. In T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones, volume 2566 of Lecture Notes in Computer Science (LNCS), pages 85–108. Springer-Verlag, 2002.

[11] Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 5505 of Lecture Notes in Computer Science (LNCS). Springer, 2009.

[12] R. Bruttomesso, A. Cimatti, A. Franz en, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT Solver. In Computer-Aided Verification (CAV), volume 5123 of Lecture Notes in Computer Science (LNCS), pages 299–303. Springer, 2008.

[13] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Operating Systems Design and Implementation (OSDI), pages 209–224. USENIX, 2008.

[14] Ned Chapin, Joanne E. Hale, Khaled Md. Kham, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. Journal of Software Maintenance: Research and Practice, 13(1):3–30, 2001.

[15] [S Chen, KG Heisler, Wei-Tek Tsai, X Chen, and E Leung. A model for assembly program maintenance. Journal of Software Maintenance: Research and Practice, 2(1):3–32, 1990.

[16] [Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. ACM Computing Surveys, 51(1), 2018.

[17] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pages 168–176. Springer, 2004.

[18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. Handbook of Model Checking. Springer, 2018.

[19] Jason Cohen, Eric Brown, Brandon DuRette, and Steven Teleki. Best kept secrets of peer code review. Smart Bear Somerville, 2006.

[20] Software Engineering Standards Committee et al. IEEE Standard for Software Maintenance. IEEE Std, pages 1219–1998, 1998.

[21] Software Freedom Conservancy. Selenium automates browsers, 2021. https://www.selenium. dev/, retrieved March 15nd, 2021.

[22] Lee Copeland. A practitioner’s guide to software test design. Artech House, 2004.

[23] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 137–148, 2009.

[24] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Principles of Program- ming Languages (POPL), pages 238–252. ACM Press, 1977.

[25] Leonardo De Moura and Nikolaj Bjorner. Z3: An Efficient SMT Solver. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pages 337–340, 2008.

[26] A. Deutsch. Static Verification of Dynamic Properties, 2003. White paper, PolySpace Technologies Inc.

[27] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/ tool-paper.pdf, 2006.

[28] D. Felsing, S. Grebing, V. Klebanov, P. Ru¨mmer, and M. Ulbrich. Automating Regression Ver- ification. In IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 349–360. ACM, 2014.

[29] Martin Fowler. Refactoring: improving the design of existing code. Addison-Wesley Professional, 2018.

[30] John E Gaffney Jr and Thomas A Durek. Software reuse — key to enhanced productivity: some quantitative models. Information and Software Technology, 31(5):258–267, 1989.

[31] Daniel Galin. Software quality assurance: from theory to implementation. Pearson education, 2004.

[32] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. Elements of Reusable Object-Oriented Software. Design Patterns. massachusetts: Addison-Wesley Publishing Company, 1995.

[33] Gerrit. Gerrit code review, 2021. https://www.gerritcodereview.com/, retrieved March 15nd, 2021.

[34] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. Queue, 10(1):20:20–20:27, 2012.

[35] B. Godlin and O. Strichman. Regression verification: proving the equivalence of similar programs. Software Testing, Verification and Reliability, 23(3):241–258, 2013.

[36] Stephan Goericke. The future of software quality assurance. Springer Nature, 2020.

[37] Carsten Gorg and Peter Weiundefinedgerber. Error Detection by Refactoring Reconstruction. In International Workshop on Mining Software Repositories, MSR 05, pages 1–5, New York, NY, USA, 2005. ACM.

[38] Mark I. Halpern. Machine independence: Its technology and economics. Communications of the ACM, 8(12):782–785, 1965.

[39] Phacility Inc. Phabricator: Discuss. plan. code. review. test., 2021. https://www.phacility.com/ phabricator/, retrieved March 15nd, 2021.

[40] ISO. ISO/IEC 14764:2006 and IEEE Std 14764-2006, 2006. Software Engineering — Software Life Cycle Processes — Maintenance. Online at: https://www.iso.org/obp/ui/#iso:std:iso-iec: 14764:ed-2:v1:en, retrieved March 15nd, 2021.

[41] ISO. IEEE/ISO/IEC 12207-2008, 2008. ISO/IEC/IEEE International Standard - Systems and software engineering – Software life cycle processes. Online at: https://standards.ieee.org/ standard/12207-2008.html, retrieved March 15nd, 2021.

[42] ISO. ISO/IEC 25010:2011, 2011. ISO/IEC Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. Online at: https://www.iso.org/standard/35733.html, retrieved March 15nd, 2021.

[43] Inc. Katalon. An all-in-one test automation solution, 2021. https://www.katalon.com/, retrieved March 15nd, 2021.

[44] Yoshio Kataoka, Michael D Ernst, William G Griswold, and David Notkin. Automated support for program refactoring using invariants. In IEEE International Conference on Software Maintenance (ICSM), pages 736–743. IEEE, 2001.

[45] Amandeep Kaur and Manpreet Kaur. Analysis of code refactoring impact on software quality. In MATEC Web of Conferences, volume 57, page 02012. EDP Sciences, 2016.

[46] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. An Empirical Study of Refactor- ing Challenges and Benefits at Microsoft. IEEE Transactions on Software Engineering, 40(7):633– 649, 2014.

[47] James C. King. Symbolic Execution and Program Testing. Communications of the ACM, 19(7):385–394, 1976.

[48] Barbara A Kitchenham, Guilherme H Travassos, Anneliese Von Mayrhauser, Frank Niessink, Nor- man F Schneidewind, Janice Singer, Shingo Takada, Risto Vehvilainen, and Hongji Yang. Towards an ontology of software maintenance. Journal of Software Maintenance: Research and Practice, 11(6):365–389, 1999.

[49] Ralf Kneuper. Software Processes and Life Cycle Models. Springer, 2018.

[50] Claude Y Laporte and Alain April. Software quality assurance. John Wiley & Sons, 2018.

[51] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Symposium on Code Generation and Optimization (CGO), pages 75–86. IEEE Computer Society, 2004.

[52] Tom Mens and Tom Tourwe. A Survey of Software Refactoring. IEEE Transactions on Software Engineering, 30(2):126–139, 2004.

[53] F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In Verified Software, Theories, Tools and Experiments (VSTTE), Lecture Notes in Computer Science (LNCS), pages 146–161. Springer, 2012.

[54] Djordje Milicevic, Mirko Brku sanin, Milena Vujosevic Janiic, Teodora Novkovic, and Petar Jovanovic. Unapredjenje programskog prevodioca Clang sa podr skom za standard MISRA/AU- TOSAR. In Etran, pages 906–910. ETRAN Society, 2019.

[55] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. Communications of the ACM, 33(12):32–44, 1990.

[56] Chahat Monga, Aman Jatain, and Deepti Gaur. Impact of quality attributes on software reusabil- ity and metrics to assess these attributes. In IEEE International Advance Computing Conference (IACC), pages 1430–1434. IEEE, 2014.

[57] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. IEEE Transactions on Software Engineering, 38(1):5–18, 2012.

[58] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. The art of software testing, volume 2. Wiley Online Library, 2004.

[59] Sam Newman. Building microservices: designing fine-grained systems. O'Reilly Media, Inc., 2015.

[60] H. Post and C. Sinz. Proving Functional Equivalence of Two AES Implementations Using Bounded Model Checking. In Software Testing, Verification and Validation (ICST), pages 31–40, 2009.

[61] Vaclav Rajlich. Software evolution and maintenance. In Future of Software Engineering Proceed- ings, FOSE 2014, pages 133–144, New York, NY, USA, 2014. ACM.

[62] Nayan B. Ruparelia. Software development lifecycle models. ACM SIGSOFT Software Engineering Notes, 35(3):8–13, 2010.

[63] Stephen R Schach. The economic impact of software reuse on maintenance. Journal of Software Maintenance: Research and Practice, 6(4):185–196, 1994.

[64] Norman F. Schneidewind. The state of software maintenance. IEEE Transactions on Software Engineering, 1(3):303–310, 1987.

[65] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why We Refactor? Confessions of GitHub Contributors. In ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pages 858–870, New York, NY, USA, 2016. ACM.

[66] SmartBear Software. Automated ui testing that covers you from device cloud to packaged apps, 2021. https://smartbear.com/product/testcomplete/, retrieved March 15nd, 2021.

[67] Mirko Spasic and Milena VujosevicJanicic. Verification supported refactoring of embedded SQL, Software Quality Journal, pages 1–37, 2020.

[68] Mirko Spasic and Milena VujosevicJanicic. GitHub repository: SQLC, 2020. https://github. com/mirkospasic/sqlc, retrieved March 15th, 2021.

[69] Ofer Strichman. Special issue: program equivalence. Formal Methods in System Design, 52(3):227– 228, 2018.

[70] Offer Strichman and Benny Godlin. Regression Verification - A Practical Way to Verify Programs. In Verified Software, Theories, Tools and Experiments (VSTTE), volume 4171 of Lecture Notes in Computer Science (LNCS), pages 496–501. Springer, 2005.

[71] M. Sutton, A. Greene, and P. Amini. Fuzzing: Brute Force Vulnerability Discovery. Pearson Education, 2007.

[72] A. Takanen, J. DeMott, and C. Miller. Fuzzing for Software Security Testing and Quality Assurance. Artech House, Inc., Norwood, MA, USA, 1 edition, 2008.

[73] Nikolai Tillmann and Jonathan Halleux. Pex – White Box Test Generation for .NET . In Tests and proofs (TAP), volume 4966 of Lecture Notes in Computer Science (LNCS), pages 134–153. Springer, 2008.

[74] Priyadarshi Tripathy and Kshirasagar Naik. Software evolution and maintenance: a practitioner's approach. John Wiley & Sons, 2014.

[75] Ervin Varga. Unraveling Software Maintenance and Evolution. Springer, 2018.

[76] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. Automated Software Eng., 10(2):203–232, 2003.

[77] Dusan Vujosevic, Ivana Kovacevic, and Milena VujosevicJanicic. The learnability of the dimensional view of data and what to do with it. Aslib Journal of Information Management, 2019.

[78] Milena Vujosevic Janicic. Concurrent Bug Finding Based on Bounded Model Checking. Interna- tional Journal of Software Engineering and Knowledge Engineering, 30(05):669–694, 2020.

[79] M. Vujosevic Janicic and V. Kuncak. Development and Evaluation of LAV: An SMT-Based Error Finding Platform. In Verified Software, Theories, Tools and Experiments (VSTTE), Lecture Notes in Computer Science (LNCS), pages 98–113. Springer, 2012.

[80] M. Vujosevic Janicic, M. Nikolic, D. Tosic, and V. Kuncak. Software verification and graph similarity for automated evaluation of students assignments. Information and Software Technology, 55(6), 2013.

[81] Milena Vujosevic Janicic. Regression verification by system LAV. InfoM — Journal of Information Technology and Multimedia Systems, 49:14–20, 2014.

[82] Milena Vujosevic Janicic and Filip Maric. Regression verification for automated evaluation of students programs. Computer Science and Information Systems, 17(1):205–228, 2020.

[83] Milena Vujosevic Janicic and Mirko Spasi c. Tools LAV and SQLAV, 2020. http://argo.matf.bg.ac.rs/?content=lav, retrieved March 15th, 2021.

[84] Peter Weißgerber and Stephan Diehl. Are refactorings less error-prone than other changes? In International workshop on Mining software repositories, pages 112–118, 2006.

[85] Peter Weißgerber and Stephan Diehl. Identifying Refactorings from Source-Code Changes. In IEEE/ACM International Conference on Automated Software Engineering (ASE), ASE 06, pages 231–240, USA, 2006. IEEE Computer Society.

[86] S. S. Yau and J. S. Collofello. Some stability measures for software maintenance. IEEE Transac- tions on Software Engineering, 6(6):545–552, 1980.

[87] Ren Yongchang, Xing Tao, Liu Zhongjing, and Chen Xiaoji. Software maintenance process model and contrastive analysis. In International Conference on Information Management, Innovation Management and Industrial Engineering, pages 169–172, USA, 2011. IEEE Computer Society.