Online ISSN 2067 - 9564 Volume 7, Number 2/2015

IMPROVING SOFTWARE QUALITY BY DESIGN

Grigore ALBEANU[1],
Florin POPENTIU-VLADICESCU[2]

Rezumat. *Această lucrare prezintă mai multe aspecte privind metodologiile recente de proiectare software, în scopul de a produce programe de înaltă calitate. Caracteristicile calitative ale programelor şi strategiile de dezvoltare software sunt factori importanti in asigurarea unui management performant al proiectelor software .*

Abstract. *This paper outlines several aspects of recent software design methodologies in order to produce high quality software. Software quality attributes and the development strategies are considered as influential factors for a successful software project management.*

Keywords: *software quality, software design, mobile apps, modern methodologies*

1. Introduction

While the engineers are able to design highly reliable hardware, the software developers are working hard to increase the quality of the code produced to be validated and installed. Some catastrophic events generated by poor software quality are well known. Starting from 1945, during the testing of Harvard Mark System, when the word "debugging" was used in computer engineering by Grace Hopper, not only catastrophic cases as: Mariner I, Therac-25, Arianne 5 and others, but also projects related to computer security motivate the engineers to study ways to improve the software quality and to manage the projects in order to maximize the system reliability under a suitable cost level [27].

According to [28], "software is developed or engineered; it is not manufactured in the classical sense", even some similarities exist - high quality being achieved through good design. In the following, the paper investigates on software quality attributes, design methodologies, and CASE tools useful to assist software development teams, and reports, in the end, some recent approaches proposed to increase software quality.

This paper is based on our previous research [1, 4, 19, 20, and 21] and describes modern software design approaches supporting reliability improvement (as required by [7] and [31]) and the current CASE tools supporting the high software quality development.

[1]Prof., PhD, Faculty of Mathematics and Informatics, "Spiru Haret" University, Bucharest, Romania, (g.albeanu.mi@spiruharet.ro).
[2]Prof., PhD, Eng., "UNESCO Chair" Department, University of Oradea, Romania, Romanian Academy of Scientists, Bucharest, Romania, (popentiu@imm.dtu.dk).

2. Modern software design approaches

According to [9], "the software life-cycle typically includes the following: requirements analysis, design, construction, testing (validation), installation, operation, maintenance, and retirement." Usually, these five phases of the Software Development Life Cycle model are used: Analysis, Design, Implementation, Testing, and Deployment and maintenance.

The analysis phase is the most important for the process of building high quality software: the problem is analyzed, the requirements are defined, and the specifications are established. The problem analysis will establish the scope of the project and a clear understanding of what the customer asks for. By interactions with customer the problem will be recursively reformulated in such a way to assure a well-defined context and to find out the answer to the following questions: *What is the context of the problem? What are the parts or elements of the problem? Why the user needs a software product to solve the problem? Who or what is affected by this problem? How the user will validate the product?*

Once a well-defined problem is given, the next activity consists of requirements' collection and analysis (to establish the product capabilities and constraints). Both qualitative and quantitative requirements have to be identified.
The requirements addresses both quality software attributes (correctness, reliability, usability, integrity, efficiency, portability, reusability, interoperability, maintainability, flexibility, testability/verifiability, survivability, expandability, robustness, stability, security, safety, and availability) and constraints about the schedule and resources (computer and input/output equipments, staff availability for operation, maintenance actions, cost for development and operation) as addressed by [4]. These requirements should be not only identified but also analyzed in order to estimate costs and benefits, and to manage the software project risk. Finally all results of the requirements collection and analyses are documented in the project plan, a reference document for all phases of project lifecycle.

From financial reason motivated by project risk analysis, the costs associated with a computer project are related to the following activities: systems analysis and design; purchase of hardware; software costs; training costs; installation costs; conversion and changeover costs; redundancy costs, and operating costs, including people costs, according to [4]. Some examples of risks are: customer will change or modify requirements, technology will not meet expectations (the advancement in technology is too fast when considers the length of the software lifecycle); inexperienced project team, delivery deadline will be tightened; users will not attend training, system will be hacked, etc.

The transformation of user-oriented requirements into software specifications is the next step. The functionality of the future software (details on input, output, process captured by the information system, and interfaces),

technical feasibility (characteristics of people and equipment required by the project development), and software quality specifications should be detailed. Not only specifications on reliability and security, but also those related to software performance and the quality of human factor should be addressed. The benefits of a good software requirements specification are: a customer-supplier complete understanding of the future product characteristics; a reduced project cost; a real estimate of cost, schedule, and risk; a clear understanding of the validation and verification steps, and improved project performability.

The phase of building the software according to the specification is called design. Following [30], the design is ‘‘the process of applying various techniques and principles for the purpose of defining a device, a process, or a system in sufficient detail to permit its physical realization.’’ Both architectural (structural) design and detailed (including algorithms) design activities will contribute to a good design. The structural design is responsible with system decomposition in order to identify, in a top-down approach, subsystems/classes and their interfaces to decrease the development time. According to [11], the phases of the design process of software products are: *data design* (to produce the data structures), *architectural design* (produces the structural units or classes, interface design (specifies the interfaces between the units), and *procedural design* (specifies the algorithms of each method). Alternative (and most used, nowadays) methodologies for the architectural design are based on the object-oriented approach. *Object-oriented systems development follows the same pattern as structured systems development*. Firstly, the system is analyzed (object-oriented analysis or OOA) and the system is designed using object-oriented design or OOD. Finally, for coding are used object oriented (OOP) programming techniques and languages.

The detailed design should provide: the software structure (data structures and algorithms or classes and the diagrams, and structure quality measurement), the software tools (languages, compilers etc.), verification/validation procedures, a test plan (items to be tested and test specifications), and the design documentation. The documented design specifications should cover, at least, the following aspects [4]: an executive summary outlining the principal aspects in the specification; a description of the proposed system and its objectives; a full description of all components (module specifications and structure charts, together with test data); descriptions of inputs (including validation tests), outputs (screen, listings, sound, etc.), and specific interfaces; a description of all data storage to include specification of file and database structure; a detailed specification of controls operating over procedures within the system, a specification of all hardware requirements and performance characteristics to be satisfied, a detailed schedule for the implementation of the system, cost estimates and constraints, and

standards to be considered for documentation, coding, testing and validation, and quality assurance.

Transforming algorithms and their data structures in computer programs, coding or implementation, in a suitable programming language is responsible for *fast, portable and bugs-free software* products. The implementation phase starts with reusable module/class/package identification (the component-based approach, the object-oriented design). It continues with code editing (writing new code, and modifying reusable code) according to a good coding style. The next step is dedicated to code inspection which includes: code reviews (mainly the aspects concerning the program logic and code readability), code quality (performance related concerns on speed, and memory used, reliability and security) and the software maintainability (how easy the code will be maintained). The final step of implementation phase is related to code testing (modules/classes/packages to be tested, testing strategies and methods, testing schedules, and the resource required for an efficient testing. A good practice is based on the test driven development methodology: test-driven development asks to developers automated unit tests, containing assertions, created before writing the code. Software testing and debugging represent some of the most important components of the software engineering process with major concerns on software reliability [27]. Even, when the newest and powerful automatic testing tools are used and a quality assurance strategy in defect removing is applied, some uncertainties in software testing can be identified in the following activities: test planning, test selection, test execution, test result checking, error tracing, and quality estimation, as stated in [17] and [18]. Software testing methods and techniques "vary greatly in variety, effectiveness, cost, need for automated tool support, and ease of use", according to [23]. Even, "program testing can be used to show the presence of bugs, but never to show their absence", as Dahl and his team shown in [8], every test performed increases intrinsic software quality by no fault reveals, or by discovering a latent fault.

The existence of specifications is essential to software testing. Correctness in software has a meaning only if the program mapping is the same as the specification mapping. Following [11], ''a program without a specification is always correct''. Therefore, the software without a specification does what is does and cannot be tested against any specification because this does violate nothing (Falsity implies anything).

Moreover, following [12], given a specification and a computer program, "any activity that exposes the program behavior violating a specification can be called testing." The efforts will be directed to test against the following classes of faults, according to [17, 18]: physical/human-made, accidental/intentional non-malicious/ intentional malicious, development/operational, internal/ external, and permanent/temporary. For web-based software, the following classes should be

taken into consideration (according to [2]): data storage class covering all possible faults related to data structures, logic faults generated during implementing algorithms and the application control flow (some of them being related to session/paging faults, inconsistent browser interaction parsing faults, mistakes in coding encoding/decoding and encryption/decryption algorithms), data input faults generated by input validation mistakes related to files and forms, appearance faults generated by inappropriate coding for controlling the display of the web-pages, and linking faults due to mistakes in controlling the transfer to different locations in the World Wide Web (URL–Uniform Resource Locator). The last class is reach for the case of web applications working with URL data bases. This taxonomy is rich enough and contains also cookies' manipulation, communication encryption, user authentication, account management, and accessing/using resources without permission.

In this view, not only program running, but also activities like: design reviews, code inspections and static analysis of source code, are labeled as testing activities even they form the so called "static testing". Following [17], the test cases design could be based on the knowledge of the code - the "white-box" method. Also, the "black-box" testing approach addressing "the overall functionality of the software is used to discover faults like incorrect or missing functions, errors in any of the communication interfaces, errors in data structures management, including databases and defects related to response time, software start-up or software termination."

According to the mentioned reference, the "white-box" testing method consists of the following tasks:

a) The investigation of all independent paths within every module, based on the flow graph analysis – "structured testing using the Cyclomatic complexity metric";
b) The examination of all logical predicates – "branch/domain testing";
c) The analysis of all loops to check their limits, and
d) The internal/external data structures validity checking.

On the other side, the "black-box" method, based on the relationships between all modules in the system model, uses the equivalence partitioning approach, the boundary-value analysis (BVA) method and, the back-to-back testing procedure. Input conditions are divided into equivalence classes (logical, ordinal values, range (interval) or set of values). The BVA approach consists of designing the test cases in order to examine the upper and lower limits of the equivalence class and the corresponding outputs. Back-to-back testing is used only for many-versions software. The versions are tested in parallel to see if the outputs are identical. Pressman & Ince, in [28], provides more insides to these testing methods. Mainly, as Pham [25] mentioned, the testing phase has the following goals: "to affirm the quality of the product by finding and eliminating

faults in the program", "to demonstrate the presence of all specified functionality in the product", and "to estimate the operational reliability of the software". Unit testing, the integration testing, and the acceptance testing are core tasks in order to obtain highly reliable software accepted by customers.

The unit test, which belongs to "white-box" approach, is the responsibility of programmers (the software house or an outsourcing entity). They have to use various approaches depending on the programming language used during coding phase. In a test-driven development approach the developers often use testing frameworks to create and automatically run sets of test cases. For many programming languages there are available unit testing frameworks. When working with components, and for some of them creating some wrappers, a regression testing is required.

The integration test includes subsystem and system test. The system test tests all the subsystems interconnected as a whole to verify functional, performance, and reliability requirements placed on major design items. The integration testing step can use the following approaches: Big Bang (all or most of the developed modules are coupled together), Bottom Up Testing (where the lowest level components are tested first, then used to facilitate the testing of higher level components), Top Down Testing (testing is conducted from main module to sub module; if the sub module is not developed a temporary program called STUB is used for simulate the component functionality), and Sandwich Testing (combine top down testing with bottom up testing).

The acceptance test is a validation of the testing phase and makes use of internal tests (in-house) and field tests (user environment). The acceptance test is defined, by Pham [25], as the "formal testing conducted to determine whether a software system satisfies its acceptance criteria and to enable the customer to determine whether the system is acceptable."

Putting into operation is the final phase of the software development process. The operation will continue with or without maintenance till the end of the software lifecycle (retirement). The main activities during this phase are: release (includes all the operations to prepare a system for assembly and transfer to the customer site), install and activate (starting up the executable component of software), adapt (modify a software system that has been previously installed) or update (replaces an earlier version of all or part of a software system with a newer release), deactivate (before adapting or updating), uninstall (before retirement), and retire (at the end of the life cycle of a software product). During maintenance the following approaches can be added: a version tracking system (to help the user find and install updates to software systems installed), and a built-in mechanism for installing updates (correcting maintenance: the modification of a software product after delivery to correct faults, to improve performance or other attributes). Other maintenance activities are of the following nature: adaptive

(dealing with changes and adapting in the software environment), perfective (accommodating to user requirements), and preventive (oriented to increase the time mission period - in general by keeping stable data structures, files and databases).

Many complementary software development methods to systems development life cycle (SDLC) there exist. However, the most important are [4]:

1. Iterative prototyping (compressing the development cycle to shorten the time to market and providing interim results to the end user) which consists, mainly, of three steps: 1) listen to customer; 2) build and revise a prototype; 3) have customer test drive the prototype and then return to step 1.
2. Rapid application development (RAD) in four phases: 1. Requirements planning - joint requirements planning (JRP), establishes high-level objectives; 2. Applications development - JAD follows JRP, involves users in workshops; 3. Systems construction - design specifications, used to develop and generate code, and 4. Cutover - users trained and the system tested.
3. Extreme programming (XP) is a new agile software development methodology adopted few years ago.
4. Other agile methods.

According to [4, 15], XP is the application of a group of practices to software development projects like:

1) The planning game: collaboration of business and programming professionals to estimate the time for short tasks;
2) Small releases: a ship cycle measured in weeks rather than years;
3) Metaphor: "a single overarching metaphor" to guide development substitutes for a formal architectural specification;
4) Simple design: no provision in the code for future changes or flexibility;
5) Testing: every piece of code exercised by unit tests when written, and the full suite of tests when integrated;
6) Refactoring: any piece of code subject to rewriting to make it simpler;
7) Pair programming: all production code jointly written by two developers;
8) Collective ownership: the right of any developer to change any piece of code on the project;
9) Continuous integration: code integrated to the main line every few Hours;
10) On-site customer: a business person dedicated to the project, and
11) Coding standards: one standard per project.

Agile methodologies are appropriate for project management processes that encourage frequent inspection and adaptation as proved in [3] and [5] for the e-Learning software domain. The mentioned references emphasize the usage of face-to-face communication over written documents (working in same location) or in different locations but having video contact daily, communicating by videoconferencing, voice, e-mail etc. It was outlined that other agile methods are

also suitable when dealing with E-learning component-based software development: Scrum, Crystal approach, Feature driven development, the rational unified process, dynamic software development, adaptive software development, open source development, Agile-CMM, Agile-CMMI etc.

3. Recent approaches in computer aided software engineering for quality improvement

The term computer-aided software engineering (CASE) is used to describe a set of tools which are able to automate all or some of various tasks of the software life cycle, like: requirements capture, tracing, and tracking; configuration management; model verification; facilitation of model validation; maintenance of all project-related information; collection and reporting of project management data, document production and CASE data import-export. Mainly, the CASE functions include analysis, design, and programming. Therefore, a CASE framework provides design editors, data dictionaries, compilers, debuggers, system building tools, etc.

There are a large variety of CASE products. A CASE product index is maintained by (http://www.unl.csi.cuny.edu/faqs/software-enginering/tools. html), belonging to a category: CASE tools (provide support for some specific tasks in the software process), CASE workbenches (supporting only one or few activities), and CASE Environments (supporting a large part of the software process).

The CASE Environments can be simple toolkits (example: the Unix Programmer's Work Bench), oriented towards a programming language (IBM Rational ClearCase), integrated (IBM Application Development Cycle), fourth generation language (4GL) based (Informix 4GL, FourGen CASE Tools: http://www.gillani.com/CASETools.htm), process-centered environments (Enterprise Architect: Enterprise II), and CASE applications (covering all aspects of the software development life cycle).

SourceForge.net provides a web-based platform for creating and publishing open software [29]. Also, GitHub offers, according to [31] “both plans for private repositories and free accounts, which are usually used to host open-source software projects”. The popularity of GitHub increased yearly, and for 2015, GitHub reports having over 10.6 million users and over 25.9 million repositories, making it the largest host of source code in the World, as GitHub press (https://github.com/about/press) announced on August 17, 2015.

The user can find a lot of tools useful to software project management, both on SourceForge and GitHub. However, software reliability is not covered, in general, by CASE products. There are specific software tools, called Computer Aided Software Reliability Engineering [27]. Many of them being dedicated to estimation of software reliability [27], such as ROBUST (Reliability of Basic and Ultrareliable Software system), FRestimate, TERSE, SREPT (Software Reliability

Estimation and Prediction Tool), SRETOOLS (AT&T Software Reliability Engineering Toolkit), SRMP (Statistical Modeling and Reliability Program), SoRel (Software Reliability Program), CASRE (Computer-Aided Software Reliability Estimation Tool), ESTM (Economic Stop Testing Model Tool), SMERFS (Statistical Modeling and Estimation of Software Reliability Functions), RGA(Software for Repairable System and Reliability Growth Analysis), DACS's GOEL (An automated version of the Goel-Okumoto NHPP Software Reliability Growth Model), according to the study realized in [6] which presents a powerful computer-aided reliability assessment tool for software based on object-oriented analysis and design, called CARATS.

SMERFS was used in [3] to study the reliability growth of two projects: a virtual campus project and the DISTeFAX software. FRestimate was used by Oradea University to teach software reliability and analyses software metrics of software modules developed by master students.

A large scale methodology is based on component-base software design. The components are developed in house or are selected from available collections. The software reliability allocation in order to obtain a high reliable integrated software product keeping cost as low as possible is applied by many organizations implementing software quality management. Software reliability optimization can be studied both in single and multi-objective frameworks. Classical optimization methods and recent approaches based on computational intelligence techniques can be used as reported in [20] for large software products over various software architectures, including fog computing extensions [21].

The increasing market of mobile applications asks also for quality assurance procedures in order to minimize risk [33], mainly due to security threats. The first step is to apply the ISO 9126 software quality model. The second step consists of risk assessment. A major challenge appears because "security risks for mobile apps go beyond those of a standard desktop application; unique concerns include the loss of a device, exposing access/data to mobile apps; employees using their devices in public/unsecured Wi-Fi hotspots; and mobile malware", according to [33]. The first category of mobile app risks addresses the "malicious functionality, which is unwanted and dangerous behaviors that are placed in a Trojan app that the user is tricked into installing". The second category addresses the vulnerabilities of mobile apps, which are bugs due to poor design or errors in implementation. In our context, the development of CASE/CASRE tools for secure mobile apps development is a must. For the moment, the software analysts and engineers use already existing CASE/CASRE tools.

4. Conclusions

This paper has described the most used software designing approaches in order to release high quality software. Various quality attributes were considered, including software reliability. The security of mobile apps is an important quality attribute and more effort should be allocated in future.

R E F E R E N C E S

[1] G. Albeanu and F. Popenţiu, *Total Quality for Software Engineering Management*, In H. Pham (ed.) "Springer Reliability Engineering Handbook" (Springer Verlag, 2003), pp. 567-584.

[2] G. Albeanu, *Agile CMMI for e-learning software development*, In I. Roceanu (coord.), R. Jugureanu, V. Stefan, V. Popescu and C. Radu (eds.), "Proceedings of the 5th International Scientific Conference eLSE 2009" (Bucharest, 2009), pp. 135-142.

[3] G. Albeanu, A. Averian and I. Duda, R & RATA, Electronic Journal of International Group on Reliability, **2**(4), 47(2009).

[4] G. Albeanu and Fl. Popenţiu-Vlădicescu, OptimumQ, **22**(3–4), 97(2011). Averian, G. Duda and G. Albeanu, *Quality assurance for agile component-based software development*, In H Pham, T Nakagawa (eds.), "Proceedings of the 15th ISSAT International Conference on Reliability and Quality in Design" (San Francisco, California, 2009), pp. 100-104.

[5] Chen et al., *CARATS: A Computer-Aided Reliability Assessment Tool for Software Based on Object-Oriented Design* (TENCON, IEEE, 2006), DOI: http://dx.doi.org/10.1109/TENCON.2006.344182

[6] D. Crowe and A. Feinberg (eds.), *Design for Reliability* (CRC Press, 2001).

[7] O.J. Dahl et al, *Structured programming* (Academic Press, London, 1972).

[8] D. Howe (ed.), *The Free On-line Dictionary of Computing*, http://foldoc.org/ (retrieved 2015)

[9] W.H. Farr and O.D. Smith, *Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS)* (NSWCDD TR 84-371, 1993).

[10] D.A. Gustafson, *Theory and Problems of Software Engineering* (McGraw-Hill, 2002).

[11] B. Hailpern and P. Santhanam, IBM Systems Journal **41**(1), 4(2002).

[12] D. Ince, An introduction to quality assurance and its implementation (McGraw-Hill, 1994).

[13] C. Jones, IBM Systems Journal **17** (1), 39(1978).

[14] K. E. Kendall, *Extreme Programming in Practice: A Human-Valued Approach to the DSI Conference Management System*, Decision Line **35** (2004)

[15] S.H. Kan, *Metrics and Models in Software Quality Engineering* (Addison Wesley, 2002).
[16] H. Madsen, P. Thyregod, B. Burtschy, G. Albeanu and F. Popentiu, *A fuzzy logic approach to software testing and debugging*, In C. Guedes Soares and E. Zio (ed.), "Safety and Reliability for Managing Risk (ESREL 2006)" (Taylor and Francisc Group, London, 2006), Vol. II, pp. 1435-1442.
[17] H. Madsen, P. Thyregod, B. Burtschy, G. Albeanu and F. Popenţiu, *International Journal of Reliability, Quality, and Safety Engineering* **13**(1), 61(2006).
[18] H. Madsen, G. Albeanu, Fl. Popentiu-Vladicescu, *International Journal of Performability Engineering* **8**(1), 67(2012).
[19] H. Madsen, G. Albeanu, Fl. Popentiu–Vladicescu and R.-D. Albu, *Optimal Reliability Allocation for Large Software Projects through Soft Computing Techniques*, In "Proceedings of PSAM 11 & ESREL 2012" (Helsinki, Finland, 2012).
[20] H. Madsen, G. Albeanu, B. Burtschy, Fl. Popentiu-Vladicescu, *Reliability in the Utility Computing Era: Towards Reliable Fog Computing*, In "Proceedings of IWSSIP" (IEEE, 2013) DOI: 10.1109/IWSSIP.2013.6623445, pp. 43-46.
[21] J.D. Meier, C. Farre, P. Bansode, S. Barber and D. Rea, Performance Testing Guidance for Web Applications (Microsoft Corporation, 2007).
[22] E. Miller, *How Software Testing Enhances Reliability*, In "Proceedings of the 5th International Symposium on Software Reliability Engineering" (IEEE, Los Alamitos, 1994) pp. 2.
[23] K. Naik, P. Tripathy, *Software Testing and Quality Assurance. Theory and Practice* (Wiley, 2008).
[24] H. Pham, *Software Reliability* (Springer, Berlin, 2000).
[25] H. Pham (ed.), *Handbook of Reliability Engineering* (Springer, 2003).
[26] Fl. Popentiu-Vladicescu, *Software Reliability Engineering* (Course book of Series of Advanced Mechatronics Systems, Debrecen, 2012).
[27] R.S. Pressman and D. Ince, *Software Engineering: A Practitioner's Approach* (McGraw Hill, 2000).
[28] SourceForge, http://sourceforge.net/ (2015)
[29] A.Taylor, The nature of creative process, In P. Smith (ed.) Creativity (Hastings House, New York, 1959).
[30] J. Tian, *Software Quality Engineering Testing, Quality Assurance, and Quantifiable Improvement* (IEEE, 2005).
[31] Wiki, *GitHub*, https://en.wikipedia.org/wiki/GitHub (retrieved August 17, 2015).
[32] ***, KPMG: Addressing mobile applications risk: A software quality focus (KPMG, Delaware, 2014).