საქართველოს მეცნიერებათა ეროვნული აკადემიის მოამბე, ტ. 16, №2, 2022
BULLETIN OF THE GEORGIAN NATIONAL ACADEMY OF SCIENCES, vol. 16, no. 2, 2022

Informatics

Robust Software Quality Assurance

Evangelos C. Papakitsos

University of West Attica, Greece

(Presented by Academy Member Ramaz Khurodze)

Software Quality Assurance is the overall activity of software evaluation, ensuring that an application meets or exceeds predetermined standards of quality. This activity is conducted in all stages of software development with the usage of inspections and testing methodologies, models and techniques. Particular and important features of software, like the number of errors and efficiency or complexity, are measured by applying related quality metrics. The preventive evaluation has a high cost of performing, although much less than having to correct errors afterwards. This paper briefly presents the relevant context and introduces the modified versions of an error monitoring model and a complexity measurement, both being well-known and most verifiable. The scope of modifications had been to considerably facilitate the evaluation process, adequately and with lower cost, as tested by the author's research teams in 20 software projects since 1992. © *2022 Bull. Georg. Natl. Acad. Sci.*

Software quality assurance, software quality metrics, software evaluation, software testing, complexity

Software Quality Assurance (SQA) consists of these processes, techniques and tools applied by professionals to ensure that software products meet or exceed predetermined standards during the Software Life Cycle (i.e., feasibility stage, analysis, design, implementation, delivery/installation, operation/maintenance and withdrawal). Without the above standards, SQA does not guarantee that any particular software product conforms to or exceeds a minimum industrially or commercially acceptable quality level. In other words, this is a very wide activity, carried out by an independent working group of programmers (different than those who have developed the examined software product), which is not involved in specific projects and submits its reports directly to the company's administration. SQA is linked to the Software Evaluation process, which is the technical part of the software quality control, while SQA is the administrative part.

The activity of SQA includes the definition of the criteria (factors), according to which the quality testing is performed, the approaches followed for the definition and usage of these criteria and finally the quality metrics or Software Quality Metrics, where quantitative recording of various important features of the software is attempted. That is, it includes methodology, standards and metrics.

Methodology

SQA planning includes the designing of inspections/reviews, metrics and evaluation of Software Development. During this process, it is defined which criteria are important in each case, while the rest are neglected. To accomplish the above task, seven specific activities are performed:

- The technical methods and tools, which the software product will be created with, are identified.
- Inspections of the Development processes are planned.
- The product is evaluated through software testing.
- The official SQA standards and procedures are enforced at all stages of software development.
- The process of software changes between the different versions of the product, in the Maintenance phase, is examined.
- Measurements of product properties are performed.
- The relevant Documentation is prepared, which includes the issuance of reports and the recording of a file for each activity of the quality inspection.

The SQA process may in some cases impede software development, due to arbitrary or inappropriate options of executing the quality inspection. The chief engineer must intervene so that SQA supports rather than hinders the production of software.

Software quality. A product of software technology consists of the software itself and its documentation. Software quality is the cause of the creation of the entire software application development methodology and other activities of the Software Life Cycle, such as the maintenance process. Of course, there are evaluation principles and techniques for each individual Life Cycle stage. It is therefore necessary to define what software quality is and how it is ensured.

Software quality is the product's compliance with [1]:

- explicitly stated operating and performance requirements;
- explicitly recorded development specifications;
- properties that are expected and implied by professionally produced software.

Then, the criteria of software quality are formulated, but for which there is no commonly accepted agreement. Thus, different views have been expressed on the composition of the set of criteria that determine the quality of software. The most famous are according to Boehm et al. [2], according to McCall [1] and FURPS (Functionality, Usability, Reliability, Performance, Supportability; see [3]), the latter created by Hewlett-Packard [4].

One way to classify the basic quality criteria is into what interests the user (i.e., Usefulness, Usability, Integrity, Efficiency, Correctness and Reliability), also called "functional", and what interests the software maintainer (i.e., Portability, Reusability and Maintainability), also called "operational". Some of the quality criteria and the factors that make them up are directly measurable (e.g., errors/KLOC), while others are indirectly measurable (e.g., Usability) through other properties and factors of the software to which they relate. Very important functional criteria are the following:

Efficiency is the amount of computing resources (processing speed, memory) required for the operation of the software.

Correctness is the degree to which the product shows correct results, according to the customer's requirements.

Reliability is whether the software provides those critical or important services that its user expects from it [2] considers Reliability as the most important criterion of software quality, which for certain reasons precedes Efficiency.

Software reviews. The design of software reviews (inspections) is a quality assurance mechanism that

includes the examination of the design and its implementation by a quality-control team, in order to find possible errors (inconsistencies, anomalies). The conclusions of the review are recorded and delivered to the chief engineer of the project, so that their repair can follow. The reviews are carried out under specific conditions, according to principles and practices, while they are divided into three categories: Production Management Reviews, Design Reviews and Formal Technical Reviews. These are conducted by creating a checklist for each case [1], i.e., they are applied in all stages of software production, following the established methodology of Software Evaluation (see next section).

An indication of the inspection cost for 100 lines of code (LOC) is two work-hours, i.e., one for preparation and one for inspection. Inspections must be short (two hours) to be effective, which means that they must be performed frequently during the software development process. The inspection rates are given as follows:

- 400 LOC per hour from the quality-control team;
- 100 LOC per hour from each member, during the preparation or the review.
- The errors discovered are classified into three categories:
- Non-critical ones, for which the cost of correction is not justified in relation to their severity, so no further action is taken.
- Repairable ones, which are presented to the developer for repair.
- Critical ones, which require redesigning the initial options.

Software Evaluation

Software Evaluation is the key activity for ensuring the quality of a software product. It is carried out through tests and inspections during the software production stages, according to the two basic principles [5]:

- To detect customer requirements.
- The tests should be planned long before they are performed, as soon as the client's requirements are determined.

Evaluation has a high cost, because the design of inspections is carried out in every component of software development, with certain specifications. However, a satisfied software user spreads his/her opinion to eight other potential customers of the specific software, while a dissatisfied user spreads his/her dissatisfaction to 22 other potential customers [3]. Therefore, the quality of software products is an important goal of software producers, despite the fact that product quality control has high costs.

If we consider as unit the cost of repairing an error that is discovered during the software design phase, then the cost of repairing the same error:

- before the evaluation tests is 6.5 times larger,
- during the tests is 15 times larger,
- after the release of the product on the market is 67 times larger.

Thus, the relationship between costs with and without preventive quality control is shaped accordingly [1]. It is observed that the total relative cost of repairing errors without preventive quality control rises to 1577 units, in relation to the 682 cost units of preventive quality control.

There are international standards for the preparation of evaluation documents developed by the Institute of Electrical and Electronic Engineering (IEEE), such as ANSI/IEEE std.830-1984 (drafting of specifications), ANSI/IEEE std.1016-1987 (drafting of design) and ANSI/IEEE std.829-1983 (drafting of review documentation).

Evaluation categories. Evaluation is divided into three categories, depending on the extent or properties in which it is carried out [6]. Adequacy Evaluation is the determination of the suitability of a system for a purpose; it is considered whether it will perform what is required of the system, how well and at what cost; significant work is needed to

identify the customer's needs. Next, during Diagnostic Evaluation, an output profile of the system is generated in relation to some possible input-field classification; it is commonly used by the project team and requires the creation of a large and representative set of test data; it also includes regression testing, where a comparison is made between successive versions of the same system. Finally, Performance Evaluation is the measurement of the performance of the system in its specific functions; criteria, metrics and evaluation methods are formulated for each function:

- Criterion: What exactly is being evaluated (e.g., accuracy, speed, error rate).
- Metrics: What property of the system is measured (e.g., speed, error rate etc.) and how.
- Method: How is the appropriate value determined for a given measurement of a tested system.

Evaluation is performed both on the individual elements of a system (intrinsic) and on the whole (extrinsic).

Evaluation processes. Software Validation is the first out of three evaluation processes, by which the developers answer the question: "Do we create the right product"? In less complex systems, Validation is performed at the analysis stage to ensure that the product created is what the user requires (i.e., "the right one"!), by checking the Specification Document for its completeness, clarity and accuracy. Thus, a detailed list is compiled of all system factors that are checked against the previous criteria. Also, correlation tables of the factors are drawn up, where it is noted whether their relationships were checked [1].

Software Verification is the second evaluation process, by which the developers answer the question: "Do we create the product correctly"? Verification is performed at the design stage to ensure that the product created is exactly what is described in the Specification Document (i.e., "correct"!). The selection of the most appropriate design method, the careful application of the design principles and the completeness, clarity and accuracy of the Design Document are the initial requirements of this process. At the discretion of the software engineer, at this stage, it is advisable to create a simple prototype of the final product ("prototyping"), which will be presented to the user, and the basic functions of the system's interface will be evaluated. The application algorithms are designed in this stage. A critical part of Evaluation is checking the suitability of the algorithms that perform the individual functions of the application. This test is based on the properties of the algorithms, for which there are both evaluation criteria and metrics. After creating the product's Specifications, the inspection evaluates whether the design implements these Specifications [1]. The evaluation of design has the following advantages:

- Errors are detected in time, before the implementation stage; this makes them cheaper to repair.
- It lasts and costs less than the evaluation of a complete program.
- There are cases (such as the use of symbolic programming languages) that is practically the only possible way of Evaluation.

The Evaluation of the design and implementation stages, which is performed through Verification, is static and mathematical. The former is applied during software development, while the latter is divided into formal (strict) and informal ("Cleanroom"):

- Formal: The commands between an input point and an output point undoubtedly lead to the expected output.
- Informal: Software defects should be avoided instead of detected and repaired; Informal Mathematical Verification is an extremely successful method, as it presents an error of 0.27% compared to 5% of the rest of the testing methodology.

The third and last Evaluation process includes a series of tests and inspections, until the final product is delivered to the user.

Quality Measurements

Conducting Formal Technical Reviews (see subsection 2.2) is based on models describing the creation and debugging of errors ("defects"), during the software development stages. The following is the presentation of the modified defect amplification model [7, 8], which is a simplified version of the defect amplification model (see [1]). The hypothesis of the modified model is described with the help of Fig. 1. At each stage of software development, defects from the previous stage enter the current one (Incoming = 10). These increase according to the Amplification Factor (= 1.5), forming the final number of defects due to the previous stages:

$$\text{Previous} = \text{Incoming} \times [\text{Amplification Factor}] = \\ = 10 \times 1.5 = 15.$$

At the current stage, additional defects are created (Current = 25), increasing their total (= 40). The inspection carried out at the current stage reveals a number of defects from the total, proportional to the Detection Factor (= 0.5), which are corrected. Thus, the defects that go to the next stage are reduced accordingly (= 20):

$$\text{Outgoing} = \text{Total} - [\text{Total} \times [\text{Detection Factor}]] = \\ = 40 - [40 \times 0.5] = 40 - 20 = 20.$$

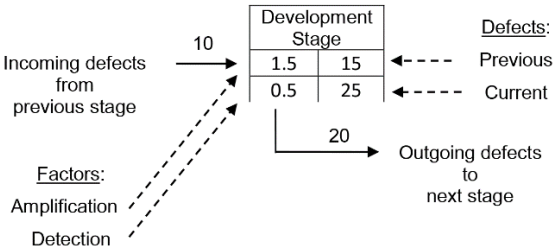

Fig. 1. The simplified defect amplification model.

The original defect amplification model [9] classifies the incoming defects in two categories: the ones that are amplified (according to the amplification factor) and those that are not. Such a distinction requires extra effort to discover them, along with the associated cost, while it was found unnecessary during the implementation of 20 small to medium-size software projects, conducted by the author and his research partners from 1992 to 2019. Therefore, the modified model (Fig. 1) has been devised for simplifying the inspection process adequately.

The values of the Amplification and Detection Factors are determined empirically. Especially for the latter, it is considered that at each stage of inspection 50% (= 0.5) of all defects can be detected. Therefore, the reliability of such models depends on the study of previous software development projects. It is therefore the result of experience, systematic collection and careful study of the relevant data. In the above general context, the software quality measurements are performed according to the quality metrics.

Quality Metrics. Software Quality Metrics are quantitative indicators that result from measurements of various factors related to the application produced. They apply to all three aspects of creating the application, namely the software itself, its documentation and production management. Therefore, they are a means of evaluating the entire production process. The effective identification and selection of metrics requires the collection of primary data during the development process, which significantly burdens the cost of production.

The measurements made are direct (cost, workhours, number of errors, number of delivered lines/commands – LOC/DSI, speed, memory size) or indirect (Complexity, Efficiency, Reliability, Maintainability etc.; see subsection 2.1). In indirect measurement, another aspect is measured, which is assumed to be related to the evaluated property, expressed in the form of a mathematical formula or model. The following measurements are an indicative presentation of the crucial property of Efficiency, for which exclusive quality metrics have been created.

Efficiency Measurements. Software Efficiency depends on the application's code. It can be measured directly (as the time of execution of the product's functions on a given machine and with the required memory) and indirectly. Indirect metrics often appear in the literature as "Software Quality Metrics" and complexity. The design of quality metrics is intended to quantify product features, preferably in an automated manner. The most important feature is considered to be the complexity of the product. The software quality and complexity metrics are more than 100. The best known and most verifiable are the following three:

- According to Gilb & Hanren, where Logical Complexity is measured, depending on the number of selection commands (Gilb) or the number of repeat commands with the software operators (Hanren). The result of this metric has been found to be proportional to the cost of the product.
- According to Halstead (Halstead's Software Science; see [10]), where various properties of software are measured, such as: the number of unique operators (=, IF, AND, < etc.), the number of unique operands (x, i, j, 3, k etc.), the total number of occurrences of operators, the total number of occurrences of operands etc.
- According to McCabe [11], where the Cyclomatic Number is calculated as a measure of the complexity of the algorithm for controlling the data flow, as well as giving a measure of the amount of required test data. From the directed graph ("flowgraph") of the algorithm (G) the Cyclomatic Number V(G) is calculated as follows:

V(G) = [number of edges] - [number of nodes] + 2

or

V(G) = [number of enclosed regions] + 1.

An example of calculating V(G) is given using the flowgraph of the algorithm in Fig. 2 [12], which includes six nodes {m, k, e, r, p, w} and 11 edges: {(m, k), (m, w), (k, e), (k, r), (e, e), (e, r), (r, w), (r, p), (p, p), (p, w), (p, e)}. Therefore, according to the first calculation method:

V(G) = 11 - 6 + 2 = 7.

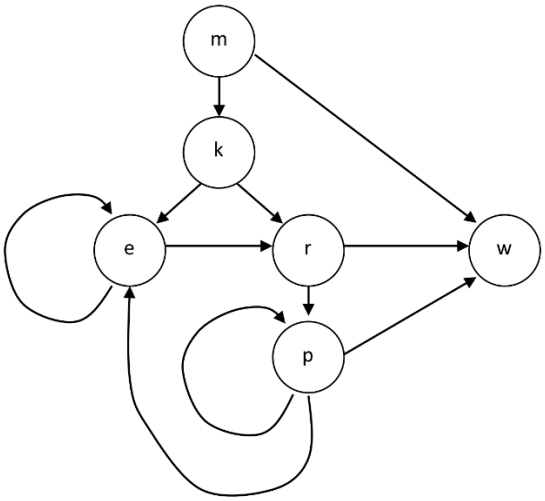

Fig. 2. The flowgraph of an algorithm.

In addition, the graph defines six enclosed regions (in curly brackets below), bounded by the following edges:

- {(m, k), (m, w), (k, r), (r, w)}
- {(e, e)}
- {(k, e), (k, r), (e, r)}
- {(r, w), (r, p), (p, w)}
- {(p, p)}
- {(e, r), (r, p), (p, p), (p, e)}.

According to the second method of calculation: V(G) = 6 + 1 = 7.

Experimental studies indicate that there is a direct correlation between cyclomatic complexity, expressed through V(G), both to the number of errors appearing in the code and in the time that takes to discover and correct them. A software module is considered complex and difficult to control when the value V(G) > 10. Finally, Cyclomatic Number V(G) is the only metric that can be calculated at the software design stage, and therefore be used proactively in product quality control. The remaining metrics are calculated a posteriori, i.e., after the implementation of the code, where it may be too late for important interventions. Therefore, only during the maintenance of software can they be useful, as well as in the accumulation of know-how for future projects.

Cyclomatic Number V(G) gives a measure of the amount of the required test data, whose purpose is to check every path of the flowgraph, namely, a piece of data for each path. The flowgraph of Fig. 2 directly depicts 11 edges from node {m} (initial/start) to node {w} (final/end). Yet, these 11 edges initially correspond to 18 different paths of the algorithm that have to be tested, while the actual paths of the particular processing problem are no more than 26, excluding edge {(p, e)} that may repeat the previous process. The total combination of paths can be huge (around 457,000), mainly due to edges {(e, e), (p, p), (p, e)} that correspond to loops. Indicatively, an algorithm of just 100 LOC, depicted by a flowgraph of 11 nodes and 15 edges that contains a single loop of 20 iterations, has approximately 100 trillion possible paths [1]. Such magnitudes are impossible to be tested. Nevertheless, each loop has to be tested for five values, namely:

- a value below the lower limit of iteration,
- the value of the lower limit of iteration,
- a value above the upper limit of iteration,
- the value of the upper limit of iteration,
- an intermediate value between the lower and upper limits of iteration.

Therefore, it is suggested herein that each loop should correspond to five regions or edges, formulating the Extended Cyclomatic Number EV(G). In this case, the flowgraph of Fig. 2, having three loops, contains 18 enclosed regions or 23 edges with EV(G) = 19, which is a value much closer than V(G) = 7 to either the 18 initial or the 26 actual paths that have to be tested, thus allowing more accurately the selection of test data.

Conclusion

SQA includes processes, techniques and criteria that are applied by software engineers for performing the quality testing, which ensures that software products conform to a minimum acceptable quality level or exceed it. Software Evaluation is the main process for ensuring the quality of software, carried out through tests and inspections during the production stages. Software Evaluation has a high cost, nevertheless, the total relative cost of repairing errors without preventive quality testing rises to 1577 units, in relation to the 682 units of preventive quality testing. Therefore, the quality of software products is an important goal of producers, despite the high cost of Software Evaluation. The Evaluation process includes a series of quality inspections through measurements and metrics.

The quality inspections are based on models that describe the creation and correction of errors. Such a model is the *defect amplification model* that classifies the incoming errors from a previous stage of software development to the current one in those amplified (according to a factor) and those that are not. Then, at the current stage additional errors are created, thus forming a total number of them. The inspection reveals a number of errors from the total that are proportional to a detection factor, which are corrected, therefore reducing accordingly the errors that go to the next stage. The amplification and detection factors are determined empirically. To avoid the additional effort and cost required for distinguishing the incoming errors in two classes, the *simplified defect amplification model* has been devised that proposes a single class of incoming errors, modifying the amplification factor accordingly.

Another crucial aspect of quality inspections is the measurement of Software Efficiency, which depends on the application's code, especially regarding the feature of software complexity. This measurement is conducted through quality metrics that are quantitative indicators describing the evaluated feature (i.e., complexity). One of the best known and most verifiable complexity metrics is the Cyclomatic Number V(G) of McCabe, which is calculated through the flowgraph of the algorithm inspected, giving a measure of the amount of required test data. V(G) is extremely important for being the only metric that can be used proactively,

since it can be calculated at the design stage of software. Yet, the calculated V(G) is not close enough to the algorithmic paths that have to be tested. Therefore, the Extended Cyclomatic Number EV(G) has been introduced herein, additionally considering the standard practices for testing loops. The value of EV(G) is much closer to the actual number of algorithmic paths to be tested.

Both modified techniques of evaluation presented herein (i.e., the simplified defect amplification model and the Extended Cyclomatic Number) have been tested by the author and his research partners during the implementation of 20 small to medium-size software projects conducted from 1992 to 2019. Their usage may facilitate the inspection process adequately.

ინფორმატიკა

პროგრამული უზრუნველყოფის ხარისხის საიმედო უზრუნველყოფა

ე. პაპაკიტსოსი

დასავლეთ ატიკის უნივერსიტეტი, ატიკა

(წარმოდგენილია აკადემიის წევრის რ. ხუროძის მიერ)

პროგრამული უზრუნველყოფის ხარისხის უზრუნველყოფა წარმოადგენს პროგრამული უზრუნველყოფის შეფასების კომპლექსურ მოქმედებას, რომელიც იძლევა იმის გარანტიას, რომ აპლიკაცია მინიმუმ, აკმაყოფილებს ხარისხის წინასწარ დადგენილ სტანდარტებს. აღნიშნულ ქმედებას მიმართავენ პროგრამული უზრუნველყოფის შემუშავების ყველა ეტაპზე და იგი ხორციელდება შემოწმებისა და ტესტირების მეთოდოლოგიების, მოდელებისა და ტექნიკის გამოყენებით. პროგრამული უზრუნველყოფის გამორჩეული და მნიშვნელოვანი მახასიათებლები, როგორიცაა შეცდომების რაოდენობა და ეფექტურობა, ანუ სირთულე, იზომება ხარისხის შესაბამისი პარამეტრების გამოყენებით. პროფილაქტიკური შემოწმება საკმაოდ დიდ თანხებთანაა დაკავშირებული, თუმცა აღნიშნული ხარჯი გაცილებით ნაკლებია შესაძლო შეცდომების გამოსწორების ხარჯებზე. წინამდებარე ნაშრომში მოკლედაა აღწერილი პრობლემატიკა, წარმოდგენილია შეცდომის სიგნალის მონიტორინგის მოდელის მოდიფიცირებული ვერსიები და სირთულის გაზომვის აღიარებული და სარწმუნო მეთოდები. მოდიფიკაციების მიზანს წარმოადგენდა შეფასების პროცესის მნიშვნელოვნად გამარტივება სათანადოდ და შედარებით ნაკლები დანახარჯით, რაც შესწავლილ იქნა ავტორის ხელმძღვანელობის ქვეშ მყოფი მკვლევართა ჯგუფების მიერ 1992 წლიდან დღემდე შესრულებული პროგრამული უზრუნველყოფის ოცი პროექტის ფარგლებში.

REFERENCES

1. Pressman R. (1987) Software Engineering: a practitioner's approach. McGraw-Hill, London.
2. Sommerville I. (1989) Software Engineering (3rd edition). Addison-Wesley, Wokingham.
3. Gialelis N.K., Dimitriadis P.D., Kalergis S.C., Kastania N.A., Katopodis K.I., Koulas R.P., Oikonomou G.T. (1999) Software applications (2nd edition). Pedagogical Institute, Athens (in Greek).
4. Grady R.B., Caswell D.L. (1987) Software Metrics: Establishing a Company-wide Program. Prentice-Hall.
5. Davis A. (1995) 201 Principles of software development. McGraw-Hill International.
6. Cole R., Mariani J., Uszkoreit H., Varile G.B., Zaenen A., Zampolli A., Zue V. (1997) Survey of the state of the art in Human Language Technology (Web Edition). Cambridge University Press & Giardin.
7. Papakitsos E.C. (2012) Quality assurance of linguistic computing products. "Technoglossia" MSc Programme in Linguistic Computing of the National & Kapodistrian University of Athens and National Technical University of Athens, Greece [in Greek].
8. Papakitsos E.C. (2013) Linguistic software engineering: I. preparation. Athens: ISBN 978-960-93-5636-7 [in Greek].
9. IBM (1981) Implementing software inspections (course notes). IBM Systems Sciences Institute, IBM Corporation.
10. Halstead M. (1977) Elements of Software Science. North Holland.
11. McCabe T. (1976) A software complexity measure. IEEE Trans. Software Engineering, 2: 308-320.
12. Papakitsos E.C. (2000) Contribution to the morphological processing of Modern Greek: functional decomposition – Cartesian Lexicon. Doctoral Thesis, Dept. of Informatics & Telecommunications, National & Kapodistrian University of Athens, Greece (in Greek).

Received April, 2022