Synoptic Coursework 2019/2020

Leandro Maglianella

Kingston University - Dependable Systems (CI6120)

Introduction

Software development requires a significant amount of effort and resources. The goal of every company should absolutely be to produce high quality software at an advantageous price compared to its competition. This result must be aimed at in such a way that, in doing so, the company is able to satisfy the customers and increase the trust they have for the company. The entire development process must be carefully designed in detail to obtain a product of excellent quality, especially when the software specifications are ambiguous or inaccurate. Two of the main activities to ensure quality are Prototyping and Testing.

Prototyping will be described in section 1.2; the Software Testing activity, on the other hand, has two main purposes: to demonstrate that the software follows the requirements and performs exactly the tasks for which it was commissioned by the client (validation testing) and to detect and correct the defects that the software may contain before deployment, through the use of artificial data prepared ad hoc to test the software (defect testing). This activity is particularly important because it represents the last phase before the release of the product and therefore, in the case in which the tests should not be precise enough, there is the risk of selling an unreliable product. A large company cannot afford such a gamble; to avoid the possibility of losing the reputation and trust of users, especially if the software released is a critical system (systems whose failure may involve serious losses, be they economic, moral or even cause people's death). Therefore the testing activity must be held in high regard, meticulously documented and must produce real evidence on software dependability. When dealing with critical systems, it is estimated that about 70% of the entire project budget is invested in the verification and validation process and that on average 80% of software bugs are detected in inspections and corrected before deployment (O'Regan, 2019). This is a good result but obviously and unfortunately does not ensure a foolproof system.

This report aims to address some particular features of prototyping and testing: in the first two sections software product metrics will be discussed, also observing how they can favour the development of socio-technical systems, in the third section the testing strategies and their application in important projects will be described, and finally in the fourth section the way in which prototyping and testing are integrated in the Extreme Programming (XP) development methodology will be studied.

1. Software product metrics and Prototyping in socio-technical systems

1.1 Software product metrics

Software product metrics are measures of quantifiable and countable software features (Stackify, 2017), whose values improve or worsen when positive traits of the functioning of the system occurs or when undesirable traits are encountered (Wadawadagi, 2017). They involve the quality of a software product (for instance, source code and design documents) (Li, 1999) and are used as predictor metrics (Sommerville, 2016). The major software features usually described by the metrics concern the system's size, complexity, design features, performance, and quality level. They help the development team to gain new insights about the efficacy of their methods and the evolution of their project (Liebed, 2018).

The objective of establishing product metrics is to be able to observe the potential of the software, to predict the quality that it will have in the future when the project will be completed and, if possible, indicate the most effective way to improve the characteristics of the product, through the identification of areas of improvement. Evaluating the status of the project can in fact discover possible risks, errors and criticalities of the software and allows you to adjust the work flow by prioritizing coherently the aspects that most need improvement. Furthermore, the metrics are a useful tool for the team manager, thanks to which a manager can evaluate the productivity of the developers and their ability to respect the development methodology adopted by the team.

Obviously, despite the importance of software product metrics and their measurement an evaluation should never be considered as an ultimatum or in any way hinder the work of the entire team: the development of high quality software must always remain the goal. Therefore it is extremely essential that the metrics are made easily collectable and analysable. The main attributes of the metrics must be simplicity and consistency, their measurement should never be ambiguous or non-objective. They must be independent of the programming language used in the development, evaluated according to invariable units of measurement and, above all, they must be relevant in order to reach the customer's specifications with the least possible use of resources.

The manager must have a good attitude and empathy in analysing the metrics, he should always consider them only as guidelines to follow and he should not obsess over them in a ruinous attempt to increase them to the bitter end, seeking perfection. Measurements should be made periodically and determined statistically after more measurements, as some metrics can quickly change over time. Finally, the metrics should never be used to evaluate the work of a particular developer (let alone firing him) but instead they should be used to understand the direction in which the whole team is generally moving. Some examples of product metrics are described below in section 2.

1.2 Prototyping in socio-technical systems

To understand how prototyping can improve the performance of socio-technical systems, it is first necessary to understand precisely what they are.

Software Prototyping consists of creating an incomplete product so that the customer can interact with it, compare it with the specifications provided and give feedback to the development team in order to achieve a better final product (Coleman & Goodwin, 2017). This activity has many positive aspects: the prototype represents the concept and the basic idea of the developers regarding the final product, allows the acquisition of a lot of useful information and provides a guide to follow. Thanks to it, it is possible to save time and money and at the same moment increase the functionality of the software: the customer and the users are engaged in the design process in a more active and meaningful way and so their knowledge of the product can advance.

Often a software company cannot simply be classified as a "technical computer-based system" but assumes the definition of "socio-technical system" (STS). This indicates a system that is not isolated and that is governed by organizational policies and rules in which all its components (be they people, machines or processes) act towards a human, social or organizational purpose. STS are generally divided into seven layers: equipment, operating system, communications and data management, application systems, business processes, organization and society (Sommerville, 2016). As can be imagined, these layers depend on each other and the correct functioning of each is fundamental to guarantee a high productivity of the system, therefore the socio-technical systems are intrinsically complex. Many characteristics are used to describe these systems, one of the most important being the dependability. It provides information on non-functional emergent properties and reflects the degree of user trust in the system. The main dependability properties are:

- Availability: the probability that the system allows the user to use its services when requested.
- Reliability: the probability that the system's output is error-free and exactly what was commissioned.
- Safety: concerns the ability of the system to work without endanger any living beings, objects or the environment.
- Security: concerns the system's ability to defend itself against possible external attacks.

It is now easy to understand how prototyping is indispensable for this type of systems: thanks to a prototype it is possible to research, test and establish more precisely what the purposes of a product are (Goyale, 2017). In doing so, early changes that save time, cost and above all increase quality can be practiced, and consequently the above-mentioned dependability properties can be boosted exponentially.

2. Four product metrics

Taking up the previous discussion regarding software product metrics, it is necessary to mention that there are many and a lot of them assume great importance in guiding the development of the project. In this section we will cover in detail only four of the most significant metrics.

2.1 Length of code

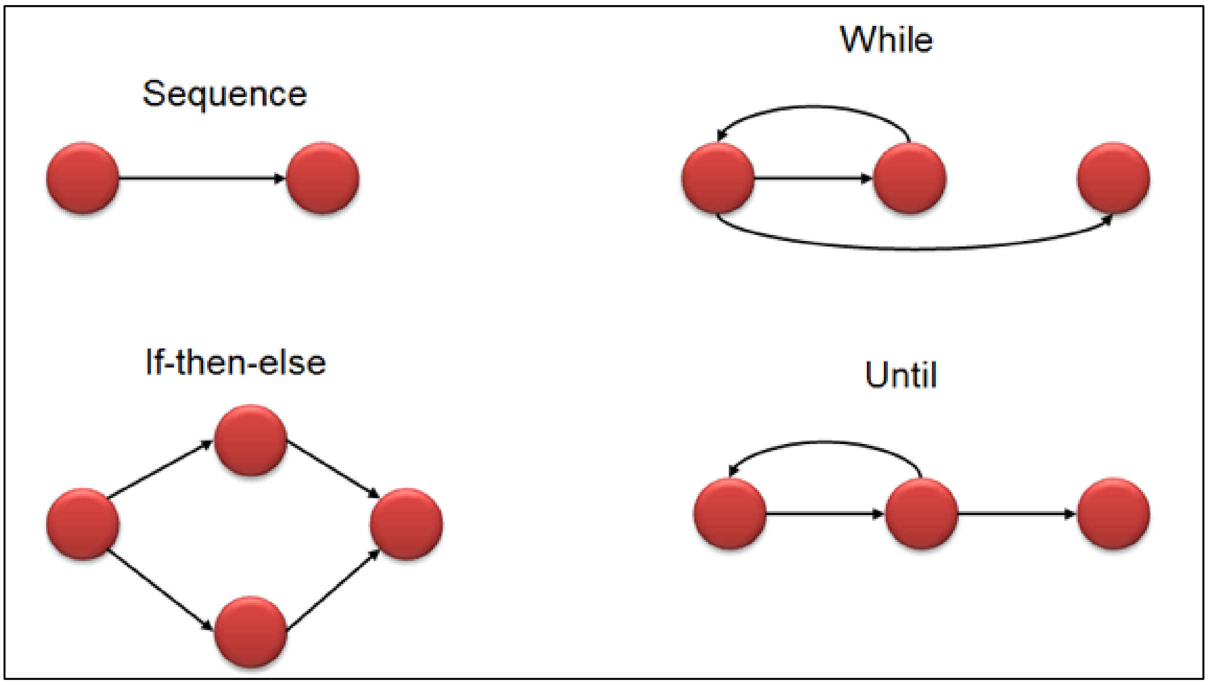
It measures the size of a program and states that the larger a code is, the more complex and error-prone it will be (Sommerville, 2016). This metric can be measured through the lines of code (LOC) and is used to predict the amount of resources needed to make the software, for example you can calculate the number of errors or the cost per KLOC (thousand LOC). However, often another unit of measurement is preferred over the LOC: the function points. Function points are a concept defined by Allan Albrecht in 1979 at IBM, they became popular starting from 1986 with the foundation of the International Function Point Users Group (IFPUG, it is the company that deals with the analysis of function points) and in 2002 they became an international ISO standard (Tutorials Point, no date). The function points quantify the functionality that a software system provides to a user and are preferable over LOC because they are independent of the programming language, objectively evaluate the codes without penalizing short but effective implementations and are measurable starting from the requirements, unlike the LOC, whose quantity can hardly be estimated before the end of the project.

2.2 Quality of comments and documentation

This metric quantifies the quality and density of the comments and documentation included in a software. Although it may seem to be a secondary and not particularly relevant measure, it should not be underestimated. Having comments in the right parts of the code can be of fundamental help when a review or change is needed later. In fact, comments represent a simple tool thanks to which a programmer can remember the logic or the reasons that led him to a particular implementation of a feature. Logically, comments should be used only where strictly required, explaining concisely the operation of the code lines to which it refers. In fact, too long or non-essential comments risk getting the opposite result, confusing and hiding the really useful sections of code. Comments in a software should be written by a programmer providing that eventually they can be useful to support other programmers who find themselves working on the same project (for instance, in the case of development teams).

2.3 Cyclomatic complexity

This metric was introduced by Thomas J. McCabe in 1976, it measures the complexity of a software and can be used to evaluate the whole code or part of it. The cyclomatic complexity is studied through a control-flow graph: the nodes represent sections of code in which the executed commands are indivisible, the direct edges connect two nodes and represent sections of the code in which the execution can take different paths depending on the evaluation of a condition. For instance, these are the graphs of the major programming statements (Guru99, 2019):



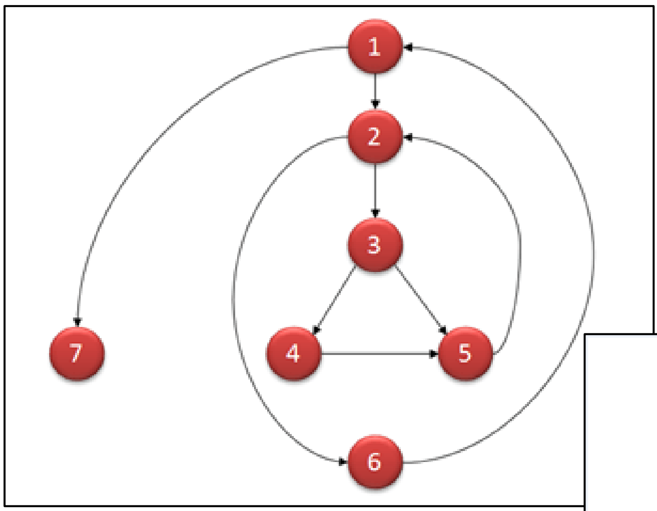
The cyclomatic complexity is defined as the number of independent paths that can be followed in the execution of the program. It is easily mathematically calculable using the following equation:

$$V(G) = E - N + 2P$$

V(G): cyclomatic complexity	E: number of edges
N: number of nodes	P: number of connected components in the graph

Generally, in a simple program the number of connected components is always equal to 1 and the equation can be rewritten in this form:

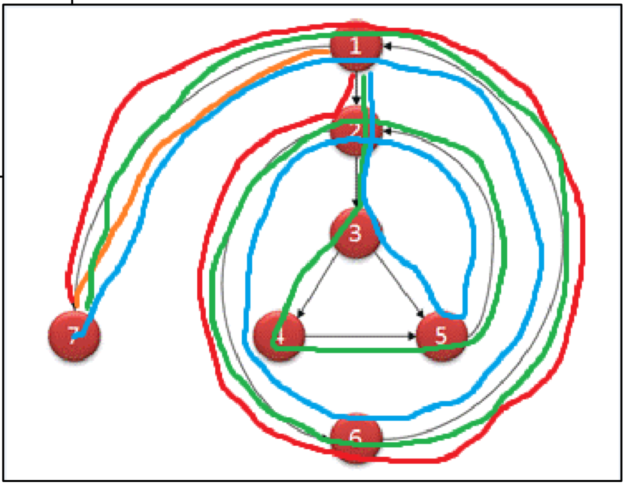
$$V(G) = E - N + 2$$



Consider for example the graph on the side (Guru99, 2019):

$$V(G) = 9 - 7 + 2 = 4$$

The higher the complexity value, the more resources will be needed to execute the code and the less the code will be testable.



2.4 Halstead complexity measures

This set of metrics were designed by Maurice Howard Halstead in 1977 through statistical studies. Using four numbers it is possible to estimate eight measures related to the complexity of the code (Metwally, 2019):

n1: number of distinct operators.
n2: number of distinct operands.
N1: total number of operators.
N2: total number of operands.

Program vocabulary:	$n = n1 + n2$
Program length:	$N = N1 + N2$
Calculated program length:	$N' = n1 \times \log_2(n1) + n2 \times \log_2(n2)$
Volume:	$V = N \times \log2(n)$
Difficulty:	$D = (n1 / 2) \times (N2 / n2)$
Effort:	$E = D \times V$
Time required to program:	$T = E / 18$ seconds
Number of delivered bugs:	$B = V / 3000$

3. Testing strategies

The purposes of testing have already been generally described in the introduction. We will now analyse the three testing phases and the strategies to which a product can be subjected before and after its deployment, to validate, verify and ensure its quality and reliability (Sommerville, 2016). All the testing strategies described below are very important and generally applicable in every type of project, starting from an individual project up to large company projects: it will simply be necessary to have the intuition to understand what is the best approach for apply them.

3.1 Development testing

It is the first phase of testing: the product is tested during its development by team members (sometimes the programmers themselves who wrote the software), the main purpose of this phase is the minimization of defects.

3.1.1 Unit testing

This process indicates the testing of the basic components of a program, such as methods, functions, classes and inheritance properties, if any. To speed up this process automated testing can be used: through a test automation framework you can extend test classes to analyse a specific code and assert whether the results produced are the same as the expected results. The goal of these tests is to understand which inputs cause errors, overflows or invalid outputs. An effective choice of inputs to test a code can be made based on two strategies:

- *Partition testing*

  Inputs with similar characteristics will be usually processed by the code in the same way, based on this the inputs can therefore be grouped in equivalence partitions. Subsequently it will be sufficient to test one or a few inputs for each partition instead of trying with many or random inputs. The most used way to identify the equivalence partitions is to find all the paths that can be followed in the execution, to understand which are the conditions that determine the existence of the various paths and finally to establish the input groups that verify or not these conditions.

- *Guideline-based testing*

  Programmers often encounter the same errors by implementing functions with similar characteristics, therefore by studying previously produced software it is possible to determine particular guidelines to follow in order not to run into such errors again. These guidelines provide important information to use in testing, such as testing a software with sequences composed of a single value or of different sizes (remembering to also use null sequences) or accessing a sequence from the beginning, middle or end.

3.1.2 Component integration testing

Many units are joined in a component and the objective of this phase is to show that the interfaces (through which the functionalities of the component are accessible) work as required by the specifications. The defects that are usually corrected in this phase concern parameters, pointers and memory areas shared between components.

3.1.3 System integration testing

Finally, the final product is made by merging all the components. All the interactions between components and the emergent properties are tested: the various components could have been developed by different programmers and therefore integrating them together could cause errors. Usually this is the most difficult test class to implement and correct, use case-based testing is one of the used approaches.

3.2 Release testing

It is the second phase of testing and concerns the validation of the system. It is necessary that the team that developed the software does not deal with it and its purpose is to check that the product meets all the customer's specifications and works effectively and dependably.

3.2.1 Requirements-based testing

Following this strategy, a tester has the task of converting specifications into tests applicable to the system and of documenting the outcome. Obviously, for this step to be feasible it is necessary that the specifications have been established in a coherent and reasonable way.

3.2.2 Scenario testing

This strategy consists of imagining and describing a particular scenario in which the software is used and subsequently derive from it tests that the program should be able to pass in order to provide the services required in the scenario.

3.3 User testing

User testing is the last phase of testing, it is necessary to definitively test the reliability of the software: it is in fact impossible for the development team to predict every case that could happen in the actual use of the product, both as regards events that are difficult to predict but also because of further interdependencies that could arise between the system and other systems already in operation. This phase begins with the dynamics of Alpha testing, in which the system is released to a restricted portion of the public, usually experienced users willing to contribute to the development. Subsequently, in Beta testing the product is released to a wider audience: in addition to allowing ever better tests, it is often also a marketing strategy that aims to make the software known to as many people as possible. Finally, in the Acceptance testing step it is decided whether the system can be definitively considered ready and whether it can be released to anyone.

3.4 Black box vs White box testing

The aforementioned testing strategies can be divided into two categories: black box testing and white box testing. In black box testing, the tester does not need computing or programming knowledge, does not know how the software is internally structured and its objective is simply to determine whether the code correctly performs the tasks for which it was written. In contrast, white box testing is usually done by software developers who know and have full access to the program code and is used to check the quality of the software. Black box testing is difficult to automate as opposed to white box testing, however black box testing is not particularly time-consuming while white box testing is.

4. Prototyping and testing in Extreme Programming

Extreme Programming is an Agile software development methodology created by Kent Beck in 1996, who also published a book in 1999 in which he describes the values, principles and practices to be applied in his approach (Hutagalung, 2006). Let us now observe how prototyping and testing (which is also one of the fundamental activities in the life cycle of the XP, together with planning, designing, coding and listening) can be integrated into this development process.

XP differs from traditional methodologies (such as Waterfall) by stating that software development must take place according to a cycle of actions that are repeated over time: the system's features must not all be ready before deployment but will be added step by step in subsequent deployments. This dynamic is imposed by practices such as "continuous integration" and "small releases". Because of this, the XP cannot scrupulously follow all the testing strategies described in the previous section, but it is nevertheless possible to ensure the validity of the software thanks to the values that the programmers should make sure to cover when writing the code. In fact, the complexity of the program should not be elevated thanks to the respect of the value of simplicity and thanks to the observance of the values of communication, feedback, courage and respect there are many opportunities, both for the client and for the team, to notice inaccuracies in software behaviour. Furthermore, the development of code through gradual additions makes it modular and easily adaptable, allowing you to easily modify it if it is necessary or if the customer decides to make changes.

The development of a system (described in the "planning game" practice) starts with the Planning phase, during this phase the customer meets the development team, provides it with the "user stories" and the requirements of the software to be produced are established. User stories are simple short sentences, usually written using "story cards", that describe how the software is supposed to behave in general. User stories must be exploited by the development team as a starting point for designing the product. Therefore a plan is established and immediately the main software production step is started: Coding. By respecting the practice of "test-driven development", Testing is performed during the coding phase, several unit and acceptance tests are created before designing the code itself, so that the programmers have the opportunity to think on which problems could cause bugs before coming across them in such a way to avoid them, effectively minimizing the number of defects. These tests are used on the software immediately after its writing and, if they are exceeded, the software is immediately released to the customer who has the task of giving feedback to the team, thanks to which it will be possible to fix any mistakes made and to approach the implementation of new features more consciously. Moreover, since each software deployment is useful to improve the product through user suggestions, then each of them can also reasonably be considered as a prototype of the final product.

Further practices are exploited by Extreme Programming with the aim of verifying, validating and assure the software, let us continue to analyse them. By "pair programming" we mean the practice which states that programmers have to work in twos on the same computer: thus doing the first programmer can focus on writing the code while the second one checks its correctness and suggests improvements, also in relation to what was previously developed. After a period of time the programmers switch roles and also exchange between other pairs to get a good knowledge of the whole project. This practice slows down the coding process but the code developed is usually of the highest quality and rarely defective. As previously mentioned, the works are directed by the customer's suggestions: however this would not be possible without the "on-site customer" practice, which established that the development activity should not be relegated to the team but instead the customer must be and feel an active member of the development, trying to be always available and ready to propose changes that must be positively embraced by the programmers. Another important aspect of the extreme programming also concerns the pursuit of the well-being of all developers: the practice "40-hour week" imposes a limit on the working hours of the team members, so that everyone is well rested and able to give his best. "Coding standards", "system metaphor", "code refactoring" and "simple design" contribute to making the programmer work in a development environment that is familiar to him, comfortable and in line with his own logic and experience (Extreme Programming: Values, Principles, and Practices, 2018). All this to accentuate a feeling of "collective code ownership" in the team through which everyone should care about the success of the project and therefore feel stimulated to write a good code. Ending, each practice of XP has as its ultimate goal the maximum increase in software quality, which is also the main purpose of Testing and Prototyping: this definitively demonstrates their rooted, albeit hidden, presence in this methodology.

Conclusion

In this report we have analysed some of the many properties concerning the two activities of Testing and Prototyping. It was possible to understand how both are characterized by precise dynamics that allow a formal and adequate implementation for a professional environment: both in fact must be applied according to a schedule and following some phases. Some measures are particularly important: these are the metrics which, thanks to their quantifiable nature, provide a simple software evaluation tool. And how can a development team make sure that these metrics are very positive and that their software is reliable and perfectly usable? Obviously by means of effective testing strategies suitable for their particular software, which should provide a path to follow in carrying out tests capable of minimizing the number of defects and capable of validating the customer's requirements. Some Agile methodologies (such as Extreme Programming) may seem to not observe this plan rigidly. However, on the other hand, through the observance of values and through practices such as test-driven development or planning games, the software is subjected to cyclical phases of development and testing that allow a lean creation of an equally reliable product and guarantee a high degree of quality in the software.

In conclusion, it is always fundamental for a developer to master these activities and be able to apply them effectively so that the code produced by him can be of unexceptionable quality.

References

- Coleman, B. and Goodwin, D. (2017) *Designing UX: prototyping.* Collingwood, Victoria: SitePoint.

- Extreme Programming: Values, Principles, and Practices. (2018) Available at: https://www.altexsoft.com/blog/business/extreme-programming-values-principles-and-practices/ (Accessed: 6 November 2019).

- Goyale, R. (2017) *Why is prototyping important?* Available at: https://blog.zipboard.co/why-is-prototyping-important-13150d76abc4 (Accessed: 2 December 2019).

- Guru99. (2019) *Mccabe's Cyclomatic Complexity: Calculate with Flow Graph (Example).* Available at: https://www.guru99.com/cyclomatic-complexity.html (Accessed: 3 December 2019).

- Hutagalung, W. (2006) Extreme Programming. Available at: http://www.umsl.edu/~sauterv/analysis/f06Papers/Hutagalung/#xp (Accessed: 5 November 2019).

- Lebied, M. (2018) *A Beginner's Guide To Finding The Product Metrics That Matter.* Available at: https://www.datapine.com/blog/product-metrics-examples/ (Accessed: 1 December 2019).

- Li, W. (1999). *Software product metrics.* IEEE Potentials, Vol.18 (5), pp.24–27.

- Metwally, A. (2019) *Calculate Halstead Complexity Measures.* Available at: https://github.com/aametwally/Halstead-Complexity-Measures (Accessed: 3 December 2019).

- O'Regan, G. (2019) *Concise Guide to Software Testing.* 1st edn. Springer.

- Sommerville, I. (2016) *Software engineering.* 10th edn. Harlow, Essex, England: Pearson.

- Stackify. (2017) *What Are Software Metrics and How Can You Track Them?* Available at: https://stackify.com/track-software-metrics/ (Accessed: 1 December 2019).

- Tutorials Point. (no date) *Albrecht's Function Point Method.* Available at: https://www.tutorialspoint.com/software_quality_management/software_quality_management_albrechts_function_point_method.htm (Accessed: 3 December 2019).

- Wadawadagi, S. (2017) *software product metrics.* Available at: https://www.slideshare.net/shreeharinw/chapter-15-software-product-metrics (Accessed: 1 December 2019).