December 2017

SQA Research

Arturas Bulavko
KINGSTON UNIVERSITY

Table of Contents

Abstract

As the software complexity increases, a systematic approach to quality must be selected to ensure it is correctly integrated into the product via the software development lifecycle. Presently, a range of software quality techniques enable to measure and predict the system's quality by applying a variety of metrics. However, due to ambiguity of quality it is not possible to systematise an approach which is applicable to most software development projects. This report focuses on reviewing several software quality models by examining the high-level quality factors to understand how each can be measured. Since the quality of the product is dependent on the process quality, various tools, validation and verification activities will be inspected. Such discoveries will be used to propose a quality management tactic for the project as well as suggest appropriate quality management tools.

1 Introduction

Software Quality does not have a true definition due to varying perspectives from the society. Some may see quality as conformance to the standards and requirements, whereas other may see quality as unquantifiable excellence which makes the product excel. Due to such perceptions, various theories and techniques were established to define and measure the quality in relation to the characteristics, commonly known as Key Quality Factors (KQF).

This report will review several quality models, discuss KQF and their metrics, as well as analyse the Software Quality Assurance (SQA) techniques. The research will begin by introducing the quality fundamentals via four dimensions of quality and differentiating the product and process quality. This will be followed by reviewing the well-known software quality models to understand their applicability towards the quality measurements. The high-level quality factors will be analysed. The report will then proceed onto reviewing various product and process metrics which can be used to assess and monitor quality throughout the development lifecycle. The research will be finished by proposing several validation and verification techniques which can be towards the projects.

2 Quality Fundamentals

2.1 Quality Dimensions

Quality is built upon four fundamental concepts, these are known as four dimensions of quality, which are: specification, design, development and conformance quality. Each will now be discussed in detail. Specification Quality denotes the extent to which the system specification is defined. Since this is the first step in the software development lifecycle, it is crucial to elicit and document the requirements correctly. By producing high-level specification, the product will be designed and developed correctly, eliminating costly changes at the later stages of the lifecycle. Chemuturi (2010, pp.26-27) suggests the following aspects when creating a product specification:

1. Functionality – specify functionality of the system.
2. Capacity – specify the system load.
3. Intended Use – specify usage scenarios.
4. Reliability – specify the operation duration before maintenance.
5. Safety – specify the safety threshold levels.
6. Security – specify the threats against which the system must be protected.

In order to correctly produce the system specification, it is necessary to engage trained and qualified personnel who will develop and follow standards setting the minimum specification boundaries. Having elicited the requirements, they must be partitioned into Functional and Non-functional requirements in order to outline the user goals and qualities of the system. Once this is done, an internal review must be performed to check whether the standards have been followed and whether the specification is comprehensive enough. The techniques for ensuring specification quality include: expert reviews, peer reviews and brainstorming. If the specification meets the quality level criteria, it is safe to process onto the next phase, otherwise the process must be completed again.

Design Quality represents the level of the design of the product which is being developed. Since design determines the shape and the advantages of the product, it is critical to do it correctly. It is common for products to fail in situations when the specification is well defined and the design is lacking. Design can be split into two phases: conceptual and engineering design. Conceptual design is the creative part, focusing on the selection of the solution from a large number of approaches. Engineering design is the details part, concentrating on working out the details for the selected approach. Both types of the design must be performed by trained and qualified personnel who must follow the specification developed earlier and the design standards. Often, to create a suitable design, the team conducts a brainstorming session to discover the possible solutions and ultimately select the most appealing one (Chemuturi, 2010, p.29). As a proof of concept, various complexity prototypes are developed, tested and evaluated to ensure the design matches the specification and standards. The techniques for ensuring design quality include: expert, managerial and peer reviews.

Development Quality refers to the method in which the system is developed. Since complete (100%) system testing is not possible due to various limitations, high emphasis is made towards the software development activities, which include: (Chemuturi, 2010, p.30)

1. Create the database and table structures.
2. Develop dynamically linked libraries for common routines.
3. Develop screens.
4. Develop reports.
5. Develop unit test plans.
6. Develop associated process routines, such as fault tolerance.

The development quality is once again achieved by qualified and trained personnel who adhere to the coding guidelines. It is also important to define and use internal standards to achieve consistency and thus positively impact on a number of KQF. In addition, the specification and design documents must be fully followed to validate and verify the product. The techniques for ensuring development quality include: peer reviews and software testing.

Conformance Quality signifies how well the Quality Control (QC) is performed in the organisation. It examines the quality at each stage using quality metrics. The tools and techniques used to measure quality include Six Sigma, ISO 9001:2008 and PQASSO (Practical Quality Assurance System for Small Organisation). The techniques for ensuring conformance to quality include: audits, benchmarking, measurements and metrics which will be discussed in this report.

While the four dimensions of quality introduce a significant quality benefit into the system, such as: improved performance, elimination of unproductive activities and producing high-quality systems, they also present a number of drawbacks. The first drawback is time because the process of staff training, process improvement and documentation generation is likely to take a lengthy period. The second drawback is cost, which is partially dependent on time. The cost of training staff, hiring of the process improvement professionals and other consultation fees are hefty. As a result of the lost time, the competitors may deploy similar systems and thus decrease the profit for the original company. Lastly, by conforming to the standards, team members are unable to innovate and develop new procedures to improve the productivity of the overall company. It is now important to understand how the four dimensions of quality are integrated into the product and process quality.

2.2 Product and Process Quality

The key attribute of a software quality is the ability for the product to deliver a specific functionality. There are two types of functionality; core and ancillary. As the name implies, core functionality refers to the critical functionality of the system which must be implemented to make it useful, primarily an aspect of specification and design. Ancillary functionality on the other hand refers to supplementary functionality of the system, implying that if it is not implemented, the product will still be used due to the core functionality. Ancillary functionality is categorised as follows: (Chemuturi, 2011, pp.36-38)

- ✓ Safety & Security – provides safety and security to the system and the users.
- ✓ Usability – enables the product to be used more conveniently.
- ✓ Fault Tolerance – product should withstand misuse without crashing.
- ✓ Feel-good – making the user feel comfortable when using the system.
- ✓ Esteem – enhances the aesthetics of the product.
- ✓ One-upmanship – enable developers to stand-out from the competitors.

By implementing the required functionality, the product can be treated as having the basic quality, however, while the product may offer all necessary functionality, it is likely to have various errors, faults and defects which impact quality. Since it is not feasible to deploy a product without defects, a permissible criterion is defined: (Chemuturi, 2011, pp.42-43)

- ✓ Critical – defects which cause failures and must be fixed immediately. The system cannot be treated as "quality product" in such case.
- ✓ Major – faults which mask the defect using a fault tolerance mechanism.
- ✓ Minor – defects which do not cause failures but reduce the quality of the system.

Through the use of such criteria, the organisation can monitor the impact on quality of their products, though, once the product is developed, the quality cannot be added to it. This results in the need to have process quality in order to ensure the quality is built-in from the start. Process quality focuses on improving the algorithm of developing a system using three steps. These steps are: process definition, improvement and stabilisation (Chemuturi, 2011, pp.200-210) which are shown in *Figure 1*.

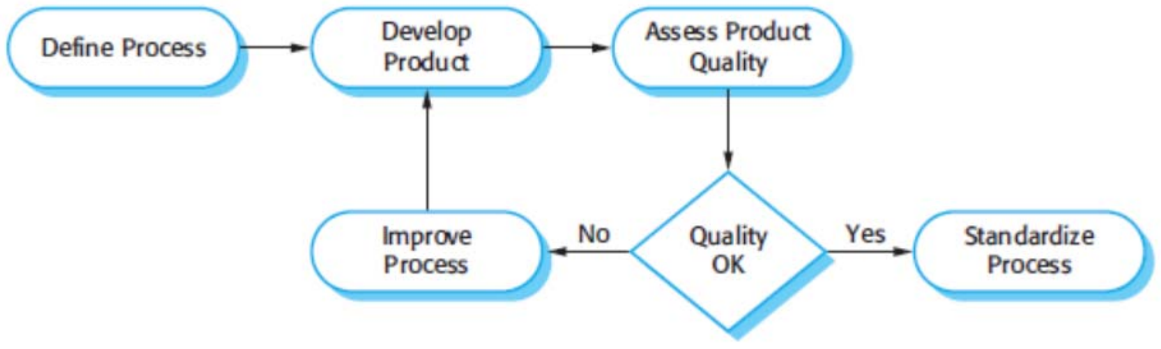


Figure 1 – Process-based Quality (Sommerville, 2010, p.657)

Process definition firstly assigns a responsible body for maintaining the definition and improving it. The definition is partitioned into top-down and bottom-up approaches. Top-down approach is mostly applicable to novel organisations which do not have a process in place. It focuses on breaking down the organisation's functions and define standards and guidelines for individual processes. Bottom-up approach is suitable for experienced organisations who wish to improve their process. It focuses on studying the current approach and adapting the best practices into the current workflow in order to improve the quality of the process. Ideally, the organisational process must be aligned with an established model such as: ISO 9000 or CMMI. Such compliance ensures best practices are utilised, which significantly improves the product quality as a result of following the process quality model. The models are reviewed in the section *3* below.

Process improvement focuses on monitoring the performance of the current approach and comparing with the desired performance, established earlier. If the process is underperforming it must be reviewed and a new technique must be adopted to ensure the targets are met. Once the processes are correctly established, meaning that the actual performance at least equates the desired performance, it is necessary to move onto the process stabilisation. Process stabilisation aids in producing predictable results which assists in planning the iterations and thus can give insights into time and budget required to finish the project using best practices and thus delivering high-quality system to the client. Essentially process is concerned with the development velocity.

By integrating product and process quality into software development lifecycle it is possible to adopt a systematic development methodology to ensure quality system delivery each time. The four dimensions of quality can also be applied towards the software development lifecycle to validate and verify the approach during all stages. Galin (2004, pp.131-136) notes that the technical complexity, extent of reusable components, qualification of team members and staff experience can negatively affect the quality assurance activities which are integrated into the development lifecycle. Such complications will impact the time and cost of the project. Having understood the product and process quality, it is necessary to examine the different types of software quality models which can be used to evaluate and measure the software requirements.

3 Software Quality Models

Software Quality Models (SQM) generate the foundation for the software quality. There are several SQMs consisting of several quality factors used to reflect the quality of the system. The models vary from large to small business and can be adapted to match the organisation's need. Such models provide a framework "for measuring software quality as a collection of desired attributes" (*Software Quality Management*, 2016). The report will now proceed onto reviewing the four commonly-used SQM which are frequently used as a starting point to develop a unique organisation's SQM as part of the process quality improvement task to improve the performance/velocity.

3.1 McCall's Quality Model

McCall's Quality Model originates from US military which established quality through a number of features. The model was developed to establish the common software quality factors to reflect the user's views and the developer's priorities (Al-Qutaish, 2010). As seen from *Figure 2* below, the features were grouped into three perspectives: Product Operation, Revision and Transition. Each feature was associated with several quality factors which could be measured using numerous reflective quality criteria by means of metrics. Product Operation is concerned with the quality of the system's operation when deployed. Product Revision looks at the maintenance and ease of change introduction into the system. Product Transition inspects the adaptability of the system towards the new environments. The eleven quality factors describe the external view of the software, while the twenty-three quality criteria focus on defining the internal view of the software. The metrics can be used to quantify and measure each criterion to ensure quality is achieved.

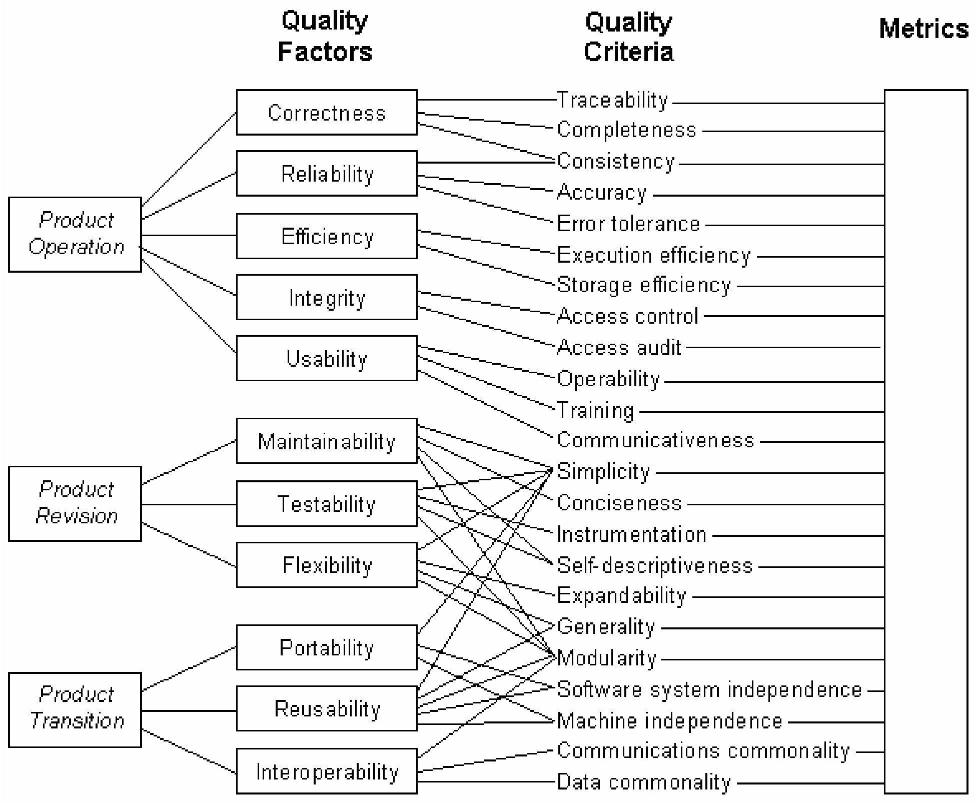


Figure 2 – McCall's Quality Model (*Software Quality Management*, 2016)

The main advantage of McCall's quality model is the visible relationships between the quality characteristics and metrics (Miguel et al., 2014). This enables to correctly quantify and measure each quality factor to ensure the right level of quality is achieved. However, McCall's model had a major drawback in terms of quality measurements. It was based on polar answers of "yes" and "no" which are is not considered very accuracy (Miguel et al., 2014). This is because some quality factors may have a larger impact on quality than the others, for example; reliability can be treated more critical than interoperability. Al-Qutaish (2010) also notes that by using polar questions to measure quality of the system, then only 50% of the quality will be achieved on that criteria. Overall, McCall's quality model forms a solid foundation for establishing a tailored model for the organisation.

3.2 Boehm's Quality Model

Boehm's Quality Model was introduced to automatically and qualitatively estimate the quality of the system (Al-Qutaish, 2010). It was advanced from McCall's quality model by introducing three aggregated high-level quality factors which are: Utility, Maintainability and Portability. These factors have a direct impact on the overall quality of the system. As seen from *Figure 3*, the high-level factors are further partitioned into seven intermediate-level factors to represent the basic high-level system requirements expected from a quality system. The low-level factors (also known as primitive characteristics) provide foundation for establishing metrics that can be used to measure the quality of the intermediate-level factors. As seen from the diagram, each intermediate-level factor can be measured by several low-level factors to ensure accuracy and correctness of the analysis.

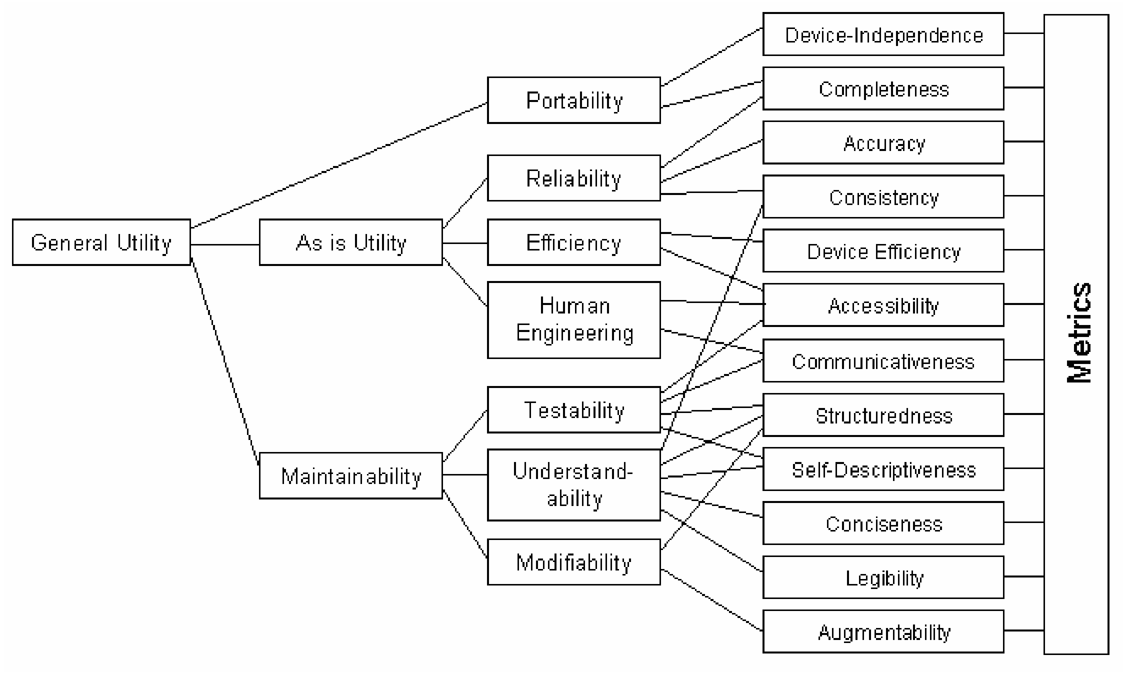


Figure 3 – Boehm's Quality Model (*Software Quality Management*, 2016)

Since Boehm's quality model is an improvement to the McCall's quality model, it is wise to examine the differences. In Boehm's model, the Testability is an element of Maintainability which suggests that if testability increases within the system, maintainability will also increase. Maintainability and testability are however separate factors in the McCall's model, implying that the system can be maintainable without being testable. Such approach on Boehm's model can be advantageous because presently, the systems are large and distributed. Therefore, by achieving high testability in such a system may positively impact on the maintainability. One of the disadvantages of Boehm's quality model is lack of decomposition criteria because it is not clear how self-descriptiveness relates to testability or how completeness impact portability (Miguel et al., 2014). Companies wishing to adapt systematic approach to quality must therefore clearly define what each factor represents and the relationship it has with the primitive characteristic.

3.3 Dromey’s Quality Model

Dromey’s Quality Model is based on “perspective of product quality” (Miguel et al., 2014). The model focuses on product quality with huge emphasis on dynamic evaluation. As a result, the model is applicable to a range of different systems (Al-Qutaish, 2010) with strict focus on relationships between the quality attributes and the sub-attributes. *Figure 4* illustrates four product qualities which can be measured using several quality attributes.

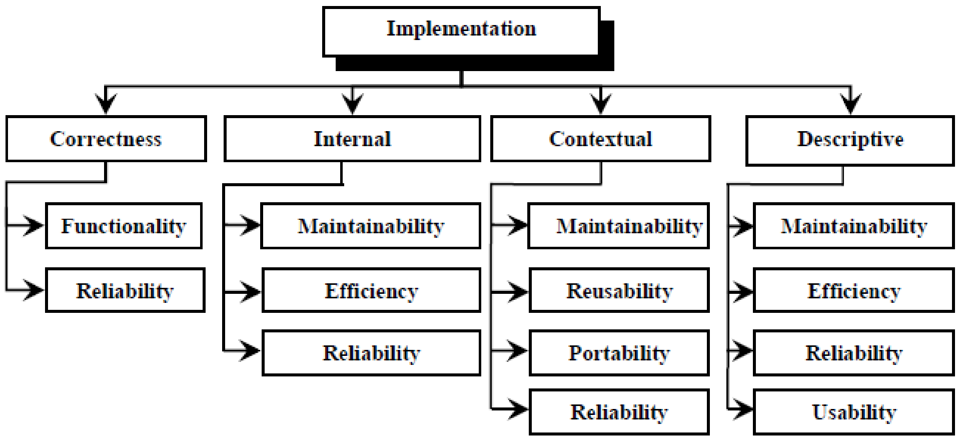


Figure 4 – Dromey’s Quality Model (Al-Qutaish, 2010)

One of the advantages of Dromey’s model is the applicability to a range of systems and products. Such diversity allows large companies to reuse same model for a range of systems without having to spend time on re-engineering their quality attributes. However, being a theoretical model, there is no guidance how the model is applicable in practice (Miguel et al., 2014). While the model can be used as basis to form new system-centric models, Dromey’s model does not provide any quantitative or measurable criteria against which the quality can be measured. Additionally, Dromey’s model does not give insights into metrics which could be applicable to measure the characteristics.

3.4 ISO 9126 Quality Model

ISO (International Organisation for Standardisation) 9126 Quality Model was formed from McCall’s and Boehm’s models in order to provide a standardised approach to defining and measuring the software product quality. ISO 9126 model is aggregated from three fundamental models. The Quality in Use aspects illustrated in *Figure 5* portrays the effectiveness of the product, satisfaction of the users as well as productivity and safety/security offered to the system (Miguel et al., 2014).

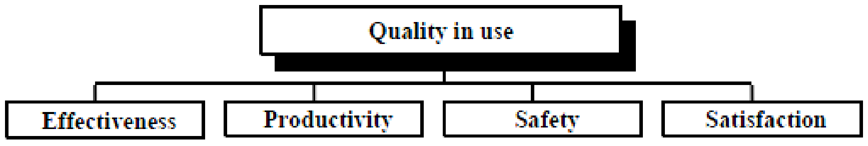


Figure 5 – ISO 9126 Quality Model for Quality in Use (Al-Qutaish, 2010)

The second model shown in *Figure 6* presents Internal and External Quality which is expected by the users during the system’s operation and maintenance. The internal quality attributes denote the system properties which can be analysed without execution, whereas external attributes represent system properties which are evaluated during the system’s execution (Miguel et al., 2014). Both, the internal and the external attributes are decomposed into six quality characteristics which are further partitioned into twenty-seven sub-characteristics. The sub-characteristics can be quantified and measured using metrics provided by ISO external and internal metrics. This enables to adapt a range of testing techniques to measure each characteristic. Additionally, ISO provide thorough definition for each quality characteristic and sub-characteristic to ensure they are perceived correctly and thus eliminate ambiguity (Al-Qutaish, 2010).

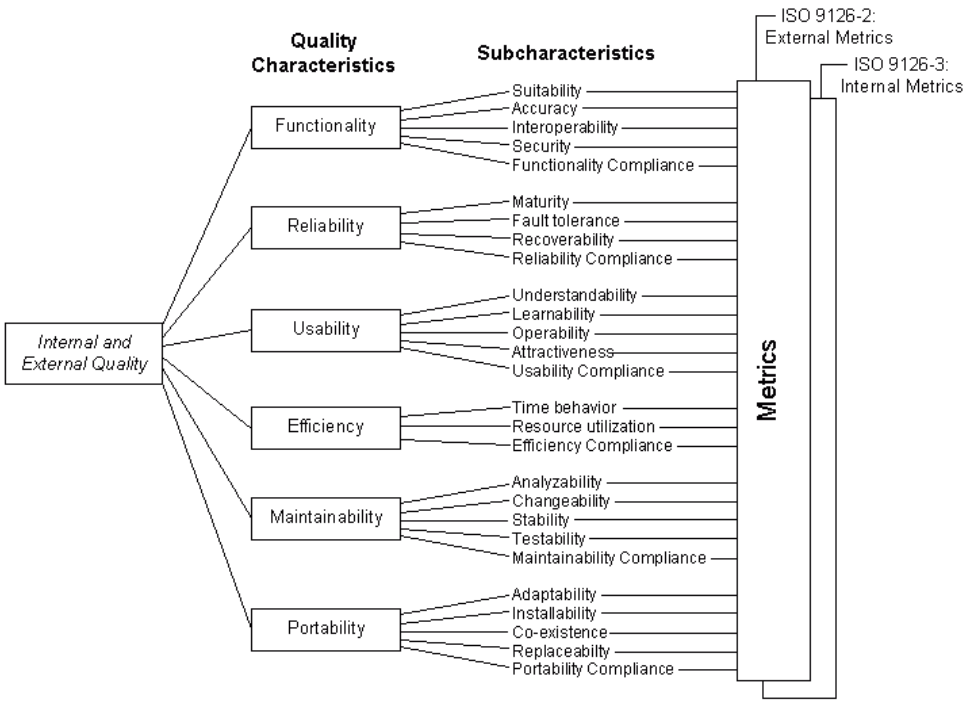


Figure 6 – ISO 9126 Model for Internal & External Quality (*Software Quality Management*, 2016)

The third model shown in *Figure 7* represents the quality in the lifecycle. It provides an overview and the expected relationships between the internal, external and quality in use attributes. From the diagram it is clear that the internal quality attributes influence the external quality attributes, which in turn influence the quality in use. Similarly, the quality in use depends on external quality attributes, which also depend on the internal quality attributes. Such relationship introduces traceability which supports in quality management at different stages.

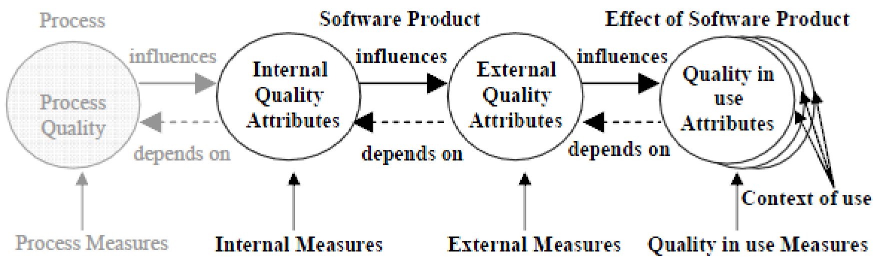


Figure 7 – ISO 9126 Quality Model in the Lifecycle (Al-Qutaish, 2010)

The major benefit of ISO 9126 quality model is due to ISO being an internationally recognised standard, meaning that the techniques and theories are widely known by a range of people working within the company, making it easier to adapt. Additionally, by providing comprehensive standards, ISO makes it easy to define and measure the quality of the product as well as process if the other ISO standards are followed (Schulmeyer, 2007, pp.67-68). As with any quality model, ISO 9126 does not prioritise the characteristics in terms of their importance to the product which is a major drawback (*Software Quality Management*, 2016). Certain factors may be more important that the others in certain systems within specific conditions, but the ISO 9126 model is not concerned about this matter.

4 High Level Quality Factors

Software Quality Models have proven that quality can be achieved, quantified and measured using Key Quality Factors (KQF). These are typically non-functional requirements (NFR) of the system. While the users are not directly concerned with NFR, they may reject the system if it does not match their expectations, for example; lacking in performance or having a poor availability/reliability. Due to this recommendation, the KQF of the reviewed models will be collected and analysed next.

FACTORS	McCALL	BOEHM	DROMEY	ISO 9126
Maintainability	✓		✓	✓
Flexibility	✓			
Testability	✓	✓		
Correctness	✓			
Efficiency	✓	✓	✓	✓
Reliability	✓	✓	✓	✓
Integrity	✓			
Usability	✓		✓	✓
Portability	✓	✓	✓	✓
Reusability	✓		✓	
Interoperability	✓			
Human Engineering		✓		
Understandability		✓		
Modifiability		✓		
Functionality			✓	✓

Table 1 – Factor Comparison Between the Four Quality Models

Table 1 presents the KQF which are reflected in the SQM reviewed earlier. Since the newer models were based on the older models, there is a KQF overlap between them. Three quality factors are present in all four quality models and two KQF are reflected in three SQM. Two factors are present in the two SQM. While the models serve as foundation for broad software systems, they do not reflect more detailed KQF which may be appropriate for a specific system. Some of these include: safety, scalability, security, robustness, adaptability, availability, modularity and complexity (Sommerville, 2010, p.656). For example, safety, availability and security are critical in medical systems where lives are dependent on a computer program. Despite this, conflicts between KQF may arise which can impact on the overall quality of the system. For instance; security affects usability, performance impacts on portability due to platform dependence (Gorton, 2011, p.37) and functionality has a negative effect on efficiency (Egyed, Grunbacher, 2004). As a result of this, it is important to consider the intended functionality of the system in order to correctly understand the quality and thus eliminate conflicting KQF to avoid trade-offs.

Based on the findings in this report, it is believed that ISO 9126 software quality model is the most applicable solution because it was based on an international agreement. Unlike McCall's model, ISO 9126 provides comprehensive quality measurements which can be used to correctly measure the quality of the system (Al-Qutaish, 2010). Additionally, ISO 9126 provides detailed metrics allowing to measure internal and external quality of the system by adopting a standardised approach. However, in order to create a high-quality system, it is necessary to define quality requirements based on the problem, rather than following a standardised approach to ensure the system correctly matches the expectations of the client.

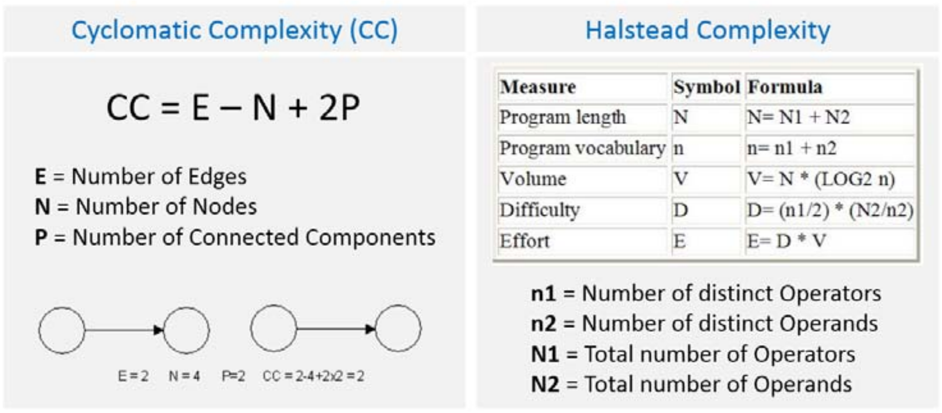
5 Software Quality Metrics

5.1 Product Metrics

The product metrics enable to assess the state of the project, uncover problematic areas, track potential risks and ultimately improve the product quality. The aim of measuring the product is to determine and predict the quality of the system. Product metrics are segregated into direct (internal attributes) and indirect (external attributes) measures. The direct measures look into the cost, effort, understanding, ease of learning, operability and communicativeness, whereas indirect measures assess the KQF. The metrics are further split into dynamic and static metrics (Sommerville, 2010, pp.672-673). The dynamic metrics are calculated during the system's execution, for example, during system testing. These metrics provide insights into efficiency, reliability KQF. The static metrics are estimated based on the plans of the system, for example, documentation and code size. These metrics aid in quantifying maintainability, understandability and complexity. This report will now review the indirect measures for the sub-factors of the most important KQF.

Maintainability KQF can be measured using a range of metrics which take into account the sub-factors. The most common metric used is Lines of Code (LOC) which looks at the whole system and counts LOC (Farid et al., 2017). This gives insights into the complexity of the system, allowing to estimate time and effort required when fixing a bug or adding new functionality. This however is a poor way of measuring maintainability because the code might not follow any programming guidelines, use meaningless names and generate long functions/methods for trivial business logic. Additionally, the code may not have any comments to support various programming decisions, reducing the maintainability of the system. It would therefore be advantageous to combine LOC with comments to form a metric allowing to assess the volume of comments per lines of code. This would enable to gain insights into maintainability from the developer's point of view.

Another way to measure maintainability is via Cyclomatic and Halstead complexity metrics (Kan, 2002, pp.314-318) which were developed to measure the module's complexity directly from the source code, shown in *Figure 8*. Cyclomatic complexity produces a single figure which in a correctly programmed system should remain below 10. Halsted on the other hand uses several operands and operators to calculate several metrics, namely; length, vocabulary, volume, difficulty and effort. It is clear that these two metrics can be used to measure other KQF such as: reliability and testability but it is suggested that each metric must focus on one KQF to keep product quality simple (Manns and Coleman, 1996, pp.95-99). While both metrics are applicable to maintainability KQF, Halstead complexity provides more metrics to completely describe the code, making it a preferred choice to the Cyclomatic complexity.



Cyclomatic Complexity (CC)

$$CC = E - N + 2P$$

E = Number of Edges
N = Number of Nodes
P = Number of Connected Components

Halstead Complexity

Measure	Symbol	Formula
Program length	N	N= N1 + N2
Program vocabulary	n	n= n1 + n2
Volume	V	V= N * (LOG2 n)
Difficulty	D	D= (n1/2) * (N2/n2)
Effort	E	E= D * V

n1 = Number of distinct Operators
n2 = Number of distinct Operands
N1 = Total number of Operators
N2 = Total number of Operands

Figure 8 – Cyclomatic & Halstead Complexity Measures

It is also possible to quantify and measure maintainability KQF by calculating the number of attributes and methods (SIZE2), depth of inheritance (DIT) and Mean Time to Change (MTTC). These metrics would also provide meaningful data to the managers about the quality of the product, but are considered low-level compared to the complexity metrics proposed earlier.

Reliability KQF can be measured by calculating the number of failures or Mean Time Between Failures (MTBF) to gain an average statistic. MTBF is applicable to repairable systems, it is also possible to use Mean Time to Failure (MTTF) towards systems which cannot be repaired. Such metrics are trivial for complex systems and thus Sommerville (2010, pp.322-323) proposes to use POFOD, ROCOF and Availability as metrics for the reliability KQF. These denote the following:

- ✓ POFOD – Probability of Failure on Demand shows the probability that a demand for a service from the system will result in failure. Such metric is applicable to situations when a failure of a system on a demand may lead to a serious problem, irrespective of the demand frequency. For example, POFOD = 0.001 means 1/1000 chance that the system will fail when a demand is made, for systems with low demand this is a relatively low probability of failure.
- ✓ ROCOF – Rate of Occurrence of Faults provides a probability of system failures within a specific timeframe. This metric is mostly suitable in situations when the demands on the system are made regularly. For example, ROCOF = 0.01 implies that there us 1 failure per 100 requests.
- ✓ Availability – This metric expresses the percentage (%) of the system delivering a service when requested. It shows that the system is operational when the demand is made. For example, AVAIL = 0.9995 shows that the system is available 99.95% of the time.

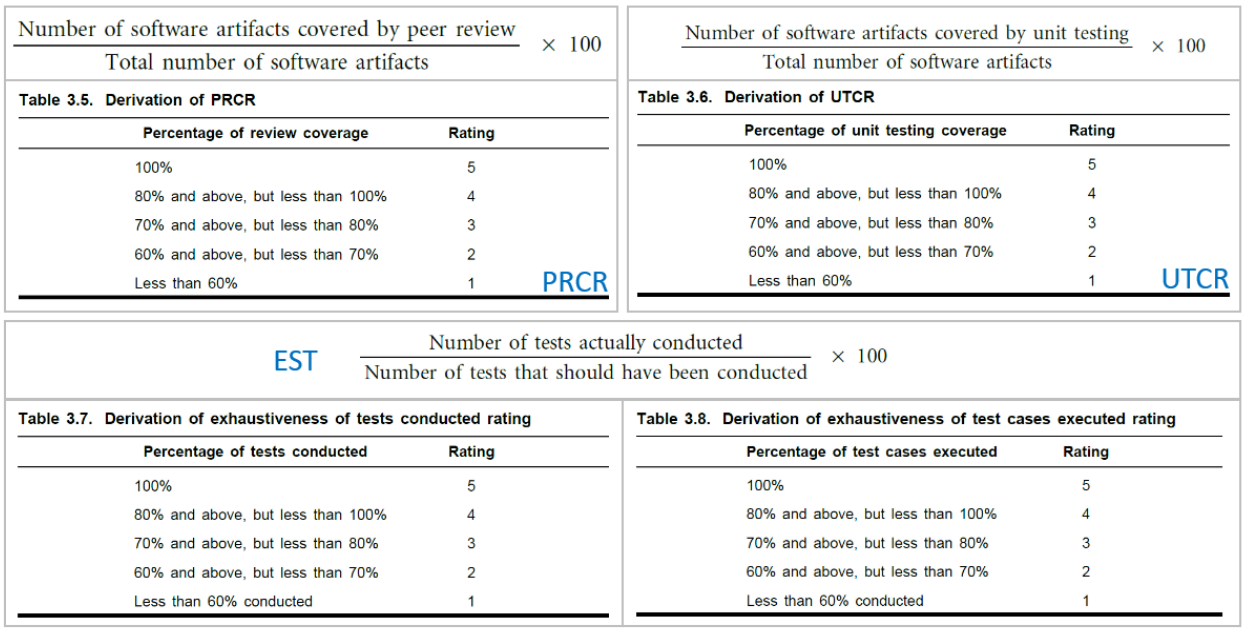
Based on the three metrics, ROCOF and Availability are fully suitable to measure reliability KQF in this project. This is because the demand on the system will be constantly made, implying that POFOD metric is not applicable in this project. Additionally, availability metric is applicable to this project because if the system is not available when the users need it, the recruitment agency will misplace customers. MTBF can also be used as a reciprocal of ROCOF to measure failure possibility.

Efficiency KQF is dependent on the programming language and the system architecture. This is due to certain languages having performance advantages as a result of their underlying architecture. Similarly, the system architecture plays a vital part in efficiency because system adopting SOA (Service Oriented Architecture) over the network may have lower latency compared to systems which run on a single machine. As a result of this, response time and throughput time metrics can be used to measure efficiency. Additionally, load and stress testing can be performed to ensure the system performs well in critical conditions. Lastly, efficiency KQF can be measured using latency time.

Testability KQF is usually measured by the amount of different tests conducted during the testing phase of the software development lifecycle. KQF can also be measured using the following metrics:

- ✓ Number of designed test cases
- ✓ Number of executed test cases
- ✓ The amount of technologies used to perform testing
- ✓ Total number of bugs/failures discovered as part of the testing
- ✓ Peer review coverage rating (PRCR)
- ✓ Unit testing coverage rating (UTCR)
- ✓ Exhaustiveness of software testing (EST)

An example of three testability metrics is presented in the *Figure 9* below.



$$\frac{\text{Number of software artifacts covered by peer review}}{\text{Total number of software artifacts}} \times 100$$

Table 3.5. Derivation of PRCR

Percentage of review coverage	Rating
100%	5
80% and above, but less than 100%	4
70% and above, but less than 80%	3
60% and above, but less than 70%	2
Less than 60%	1

PRCR

$$\frac{\text{Number of software artifacts covered by unit testing}}{\text{Total number of software artifacts}} \times 100$$

Table 3.6. Derivation of UTCR

Percentage of unit testing coverage	Rating
100%	5
80% and above, but less than 100%	4
70% and above, but less than 80%	3
60% and above, but less than 70%	2
Less than 60%	1

UTCR

EST $$\frac{\text{Number of tests actually conducted}}{\text{Number of tests that should have been conducted}} \times 100$$

Table 3.7. Derivation of exhaustiveness of tests conducted rating

Percentage of tests conducted	Rating
100%	5
80% and above, but less than 100%	4
70% and above, but less than 80%	3
60% and above, but less than 70%	2
Less than 60% conducted	1

Table 3.8. Derivation of exhaustiveness of test cases executed rating

Percentage of test cases executed	Rating
100%	5
80% and above, but less than 100%	4
70% and above, but less than 80%	3
60% and above, but less than 70%	2
Less than 60% conducted	1

Figure 9 – Sample metrics for Testability KQF

As seen from *Figure 9*, the testability KQF are based on rating, higher probability results in a higher rating which provides a quantified measurement for the KQF. Ming-Chang (2014) also notes that readability metrics can be applied towards testability. It is believed that highly-readable code will be easier to test using $D = 49L^{1.01}$ formula, where D is the number of pages in the document and L is the number of 1000 lines of code (for a system with 40,000 lines of code, L will be 40). Lastly, testability KQF has numerous tests to ensure system correctness which is covered in section *2.6.2* of the report.

Portability KQF can be measured by calculating the probability of the devices that the system is able to run on. This includes different operating systems, device brands and hardware configurations. Additionally, it is possible to calculate the time taken to run/install the software by an average user in the regular environment. Furthermore, be comparing the software architectures it is possible to estimate how portable the system will be by adopting a specific programming paradigm. For instance, system developed using SOA can be seen as more portable when compared to a system developed in the old-fashioned three-tier architecture.

Usability KQF can be measured using customer satisfaction level or more correctly using customer problem metric (Ming-Chang, 2014). This is frequently calculated using customer-founded defects total (CFD) by adopting the formula presented in *Figure 10* below.

$$CFD\ total = \frac{Number\ of\ customer - founded\ defects}{Assembly - equivalent\ total\ source\ size}$$

Figure 10 – CFD Formula (Ming-Chang, 2014)

The formula outputs a quantifiable quality attribute which can be used to assess the quality of the product. Additionally, it is possible to calculate the average time it takes for a customer to complete their tasks or the amount of time required to train one user to use the system. Sommerville (2010, p.672-673) proposes to use Fog Index in order to acknowledge how difficult the user manual is to understand. Fog index focuses on measuring the length of words in the documents which can be used to estimate how complex the user manual is. This would enable to quantify usability because if the user manual is simple enough for people with varying computer literacy levels to understand, then customers are likely to learn the system functionality by reading the user manual.

The metrics discussed in this section are beneficial towards KQF because they enable to quantify and accurately measure each factor by evaluating its sub-factors. Such systematic strategy allows to improve quality of the product as well as management knowledge for the future projects. Metrics also have a number of drawbacks. Firstly, there are countless metrics which could be used to measure a single KQF, it would be useful to develop a framework with the best metrics to use for each factor and sub-factor, because presently it is a matter of choice due to lack of standardisation. Secondly, certain metrics consume great amount of time, which in small organisations is problematic. This implies that such organisations should either adopt automatic metrics or focus on less KQF when developing a system. Such approach may negatively impact on the overall product quality. Lastly, due to such a comprehensive quality assurance (QA), team members are likely to fake results from metrics to decrease their workload (Galin, 2003). In practice, not all developers are concerned with the quality of the product and thus can manipulate the metric results to product excellent reports. This however can be avoided through process quality assurance metrics which will be discussed next.

5.2 Process Metrics

The process metrics provide insights into software engineering tasks, milestones, process paradigms, ultimately leading to the process improvement and the development of team's productivity. Additionally, process metrics give insights into efficiency of the tools used which in combination can provide project schedule estimates. By applying metrics towards the process, it is possible to assess whether the activity can be improved, how individuals may become more productive and average time required to complete a specific activity. This aids in planning and estimating upcoming products.

Sommerville (2010, pp.711-714) partitions process metrics into three main types:

1. The length of a specific process. This can be the total time spent on a process, time dedicated to a process or process completion time by a particular developer.
2. Process resource requirements. Resources can include costs, effort in person-days and hardware which will be used to develop the product/system.
3. The amount of event occurrences. May include the average amount of discovered defects during testing or the number of requested requirement changes.

By analysing types 1 and 2 it is possible to assess whether the process changes improved the efficiency of the project. This enables to examine the current approach and introduce novel strategies to improve the efficiency of the process. Such strategy enables to measure the process at the fixed points and present improvements which will positively impact the product. Type 3 is focused towards product quality because by changing process approaches it is possible to improve the quality of the product. For example, by optimising the program inspection process, the product is likely to have better quality characteristics. The main difficulty of process measurement arises during the data capturing, because there is no concrete answer as to what needs to be measured. The GQM (Goal Question Metric) paradigm shown in *Figure 11* was developed to measure the process.

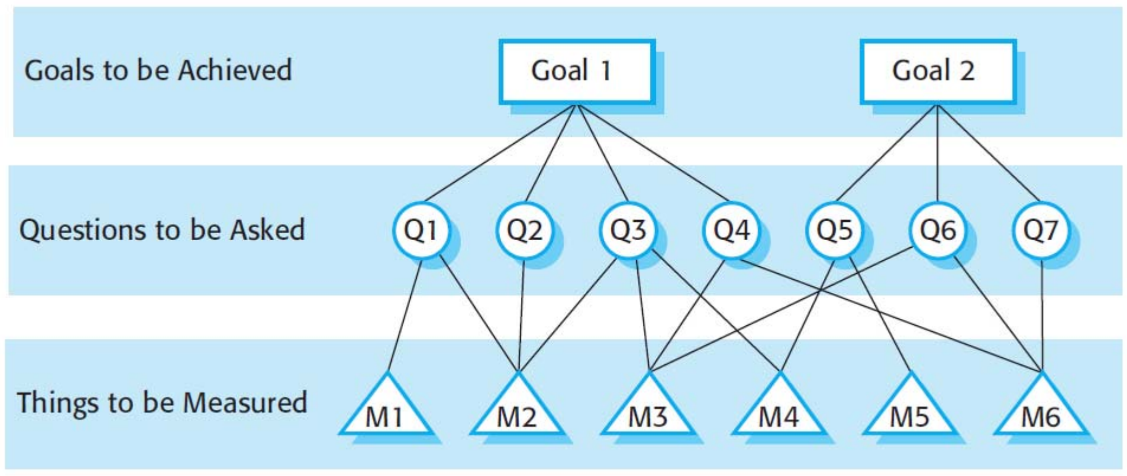


Figure 11 – GQM Paradigm (Sommerville, 2010, p.712)

The GQM paradigm is typically used as a process improvement technique which partitions the assessment into three abstractions: (Sommerville, 2010, pp.712-713)

1. Goals – This reflects the organisation's aims which they are trying to achieve.
2. Questions – These are the business' objectives which reflect the detailed aim decomposition.
3. Metrics – Measurements which must be collected in order to confirm the effectiveness of the process, providing qualitative analysis of the current approach.

The primary advantage of GQM paradigm is separation of concerns. The goals (organisational concerns) are not directly related to the questions (process concerns) due to their level. Fundamentally, the paradigm aids in selection of data that be collected and how it can be analysed to provide meaningful and correct feedback to the managers. The appropriate metrics can then be chosen to measure each question. The GQM paradigm follows similar pattern to software quality models and thus the disadvantages are similar. It is not clear what criteria should the metrics be based on, in other words, there is no standardised method to measure specific element. By selecting incorrect metric, the results might be revealing that the process is at the peak of productivity, but the product is lacking in quality. There is also no prioritisation of goals which may seem as a disadvantage.

As part of the process metrics it is possible to quantify the quality via defect detection and removal effort. Each organisation must have a document with defect classification scheme which contains several attributes of each defect, it may include: (Manns and Coleman, 1996, pp.109-115)

- ✓ Defect severity and symptoms
- ✓ Where, when and how found
- ✓ Where, when and how caused
- ✓ Where, when and how fixed
- ✓ Resources required to fix the defect

Such information will enable to estimate the average cost and effort of resolving defects within a specific category. This also enables to estimate the resources required for the project and thus attempt to introduce improved quality techniques. For example, dedicating more time towards testing phase of the development lifecycle and applying a range of testing methods to improve the quality of the product. Sommerville (2010, p.668) notes that process quality can be measured by calculating the "average effort and time required to repair reported defects". This can be partially achieved by applying the earlier discussed metrics. It is also possible to measure effectiveness using the formula shown in *Figure 12*. By measuring the effectiveness of individual activities, it is possible assess and improve the process to achieve greater efficiency (Schulmeyer, 2007, pp.410-411). Galin (2004, pp.419-420) also proses to use software process timetable metrics to assess achievements and failures in one sprint or iteration. The formulas presented in *Figure 12* provide Time Table Observance (TTO) and Average Delay of Milestone Completion (ADMC) will quantify each milestone and provide statistics about the problems which arose, allowing to correct them and thus improve process quality.

Effectives Measure	Software Process Timetable Metrics
$E = \frac{N}{N + S}$	$ADMC = \frac{TCDAM}{MS}$ $TTO = \frac{MSOT}{MS}$
E – Effectiveness of activity **N** – number of found faults found by activity **S** – number of faults found by subsequent activities	**MSOT** – Milestones completed on time **MS** – Total number of milestones **TCDAM** – Total completion Delays for all Milestones
Defect Density (DD)	Review Efficiency (RE)
$DD = \frac{\text{Total Number of Defects}}{\text{Project size in KLOC or FP}}$	$RE = \frac{\text{Number of defects caught in review}}{\text{Total number of defects caught}} \times 100$

Figure 12 – Process Metrics

Defect Density and Review Efficiency metrics illustrated in *Figure 12* can also be used to measure the process quality. KLOC stands for kilo lines of code and FP denotes function points. The review efficiency will provide insights into the performance of conducting product reviews as part of product quality improvement. By assessing such aspect, it could be possible to optimise specific aspects of the process or apply different tactics to ensuring product quality. Overall, by improving the process quality, the quality of the product improves given that the techniques are applied correctly and by trained staff. If done incorrectly, the product may not be finished or it may even contain greater number of defects due to time spent on producing documentation. It is also important that all members of the organisation are interested in improving the process by following specific paradigms. It is also worth noting that during process improvement, it is not advised to pick on people which are responsible for problems. The manager should rather analyse why the problem has happened and how it can be rectified next time in order to avoid same issues.

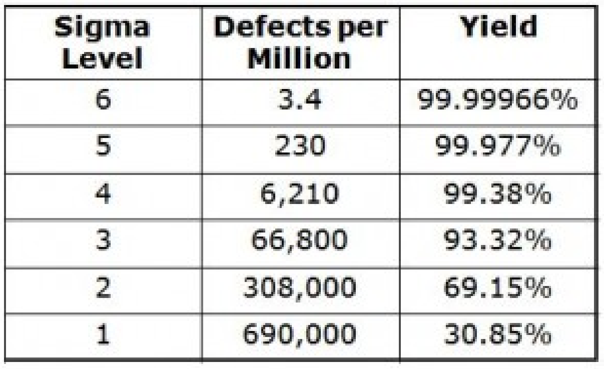
Six Sigma provides a combination of tools and techniques for process improvement by following a rigorous validation process. Six Sigma can be seen as a customer-focused change strategy allowing to improve the organisation's performance using a systematic approach. It enables to measure quality for organisations wishing to eliminate defects and thus achieve the highest level of excellence. In order to achieve this, Six Sigma has six key values/concepts: (Desai, 2010, pp.9-11)

- ✓ Focus on the Customer – the product must satisfy the customer requirements.
- ✓ Data and Fact-driven Management – Using metrics to assess and analyse the performance.
- ✓ Focus on Process, Management and Improvement – Measure present performance and improve efficiency in order to increase quality which will satisfy more customers.
- ✓ Proactive Management – Establishing clear goals and adopting new technology.
- ✓ Teamwork – All members of the team are treated equally.
- ✓ Drive for Perfection, Tolerate Failure – Promoting risk management and removing defects in order to deliver high quality products developed as part of high quality process.

By focusing on these six values, Six Sigma proposes the following five steps to process improvement which directly affect the product: (Desai, 2010, pp.20-21)

1. Define – describing the problem in as much detail as possible, include relevant diagrams.
2. Measure – assessing the current efficiency of the process,
3. Analyse – the application of statistical analysis or metrics to find root cause of the defects.
4. Improve – elimination of root cause of the problems and finding a suitable approach.
5. Control – establishing the improved process into the development lifecycle.

Once the five process improvement steps have been undertaken, it is possible to obtain the Sigma level, which is basic six standard deviations. As shown in *Figure 13*, the defects are significantly reduced per level, suggesting that 6 Sigma level outputs the best quality product.



Sigma Level	Defects per Million	Yield
6	3.4	99.99966%
5	230	99.977%
4	6,210	99.38%
3	66,800	93.32%
2	308,000	69.15%
1	690,000	30.85%

Figure 13 – Levels of Six Sigma Performance (Learn Six Sigma Definition, 2018)

The primary advantage of Six Sigma when compared to traditional TQM (Total Quality Management) is that it focuses on value by reducing the amount of defects, whereas TQM examines quality as conformance to the requirements (Desai, 2010, p.14). Similarly, Six Sigma is aimed at optimising all business processes while TQM focuses on improving individual operations within the process. Another advantage of Six Sigma is pushing individuals towards achieving better quality. One of the disadvantages of Six Sigma is the lack of motivation of employees to adopt the model (Barjaktarovic and Jecmenica, 2011). It is not possible for all the company employees to appreciate the quality improvement steps which builds pressure on each individual. It is also arguable whether elimination of defects positively affects the development lifecycle, because by focusing on quality some important user functionality could be neglected. However, various researchers are keen on integrating Six Sigma with the software development lifecycle to improve the system quality (Kwak and Anbari, 2006). There are several alternatives to Six Sigma, famous examples include: CMMI (Capability Maturity Model Integration), SPC (Statistical Process Control) and Engineering Process Control (EPC). The models share some characteristics, enabling the company to select the most suitable one to improve their quality.

6 Software Verification & Validation

6.1 Software Verification

The aim of software verification is to confirm that the system conforms to the specification, regulation or standard. It is primarily a manual process performed by visual means to ensure best practices have been followed and thus ensure quality. Verification enables to find and correct defects as soon as possible, eliminating the need of reworks and fault corrections during the later stages of the project when the costs might increase. Techniques such as walkthroughs, inspections and audits are performed by qualified individuals to ensure quality is always achieved. The main advantage of verification is the lack of the need to have a fully working product because the assessor can just look at the structure of the document/code to identify potential defect (Manns and Coleman, 1996, p.126).

Walkthroughs are also known as peer reviews with the notion of people other than the author to go through the artefacts in order to discover problems. The goal of the walkthrough is to evaluate the artefacts, examine the validity of the approach and achieve a common understanding, allowing to exchange feedback. It is important that the person performing walkthrough has some background knowledge similar to the author in order to ensure high level of quality is achieved due to understanding. Walkthroughs are typically split into five types: (Chemuturi, 2011, pp.90-101)

- ✓ **Independent** – the author is not present during the walkthrough. This enables the author to spend time developing another artefact, however, if the reviewer does not understand the system to its complexity then the walkthrough will output meaningless data.
- ✓ **Guided** – the author guides the reviewer through the artefact. This saves valuable time for both parties, however, they may both disagree about the defect and thus a conflict will arise.
- ✓ **Group** – more than one reviewer is required to review the artefact. Postal, meeting and guided meeting review models can be adopted to conduct a group review. Such approach enables to find a consensus, but organising such a meeting could be hard in a busy company.
- ✓ **Expert** – internal or an external reviewer who is qualified in the domain of the artefact is asked to conduct a walkthrough due to absence of the required people within the team. Such type of walkthrough is costly, nonetheless they may be the only source of assessing the product in terms of quality when a novel technology is used.
- ✓ **Managerial** – performed by the supervisor of the author. This may take the form of independent or guided walkthrough to ensure the system works correctly. Such review is not primarily concerned with defect discovery.

Each type of the walkthrough provides a report documenting participants, examines artefacts, walkthrough objective, a list of identified issues and recommendations that were made (Manns and Coleman, 1996, p.139). Such approach introduces traceability into the process allowing to measure it. Chemuturi (2011, p.102) notes that group and expert walkthroughs are very unlikely to be performed in a typical software development company due to time limit and high cost. As a result, independent, guided and managerial walkthroughs have a direct impact on the quality, meaning that the reviewer must be qualified to correctly evaluate the artefact. Agile Extreme Programming (XP) saw the benefit of walkthroughs/reviews and has therefore introduced pair programming practice to ensure quality of the product is developed along the process.

Inspections ensure that all necessary artefacts are ready for the next phase. The aim of inspection is to find a record defects (Manns and Coleman, 1996, p.136). It is typically performed by three to six people to ensure the team is easily manageable. Individuals represent a particular role to reflect an activity during an inspection, namely; author, moderator, reader, recorder and the inspector. The inspections are split into the following types: system testing readiness, acceptance testing readiness and delivery readiness inspections (Chemuturi, 2011, pp.102-110). As the name suggests, system testing readiness inspection ensures that the artefact is ready for a formal system testing. This ensures that all relevant components have been developed and testing strategy has been devised. Acceptance testing readiness inspection ensures the artefact is ready to be shown to the customer. The inspection confirms that the system is ready to be tested by the customer, all documentation is ready and an approved acceptance plan is developed. The delivery readiness inspection checks whether the final product contains all necessary documentation, all QA activities are carried-out and all critical defects are eliminated. Most organisations perform inspections to ensure all critical activities are conducted smoothly. However, some organisations argue that inspections are not needed due to phase-end audits. This is incorrect because inspections also assess the quality of the documents and the system itself to ensure correctness and readiness.

Audit is a document verification process to ensure it matches the organisation's standards and the defined process. The aim of an audit is to uncover non-conformances within two hours which are recorded in a non-conformance report as a result of the audit process (Chemuturi, 2011, p.110). There are several varieties of audits: (Chemuturi, 2011, pp.113-123)

- ✓ Conformance – measures the degree towards which the documents conform to standards.
- ✓ Investigative – discovers the causes of failures and extraordinary success.
- ✓ Vertical – conducted across the entire organisation to ensure conformance in all domains.
- ✓ Horizontal – focus on conformance to the standards by assessing one project.
- ✓ Periodic – conducted during the scheduled intervals.
- ✓ Phase-end – triggered by project events and are spontaneously conducted.
- ✓ External – conducted by an external auditor to ensure compliance with standards such as ISO.
- ✓ Internal – conducted by the organisation's internal staff.

Each type of audit serves a major benefit towards the quality of the process and product because the conformance is assessed at all stages. Chemuturi (2011, pp.123-124) indicates that audits are a preferred way when compared to inspections because the process is more systematic and less time-consuming. By having the total control of the process via audits it is possible to assess and improve the quality using the metrics proposed earlier.

Verification has several advantages. Firstly, productivity is increased because errors are eliminated early in the project and thus the quality of the software is improved (Manns and Coleman, 1996, p.127). Secondly, the maintainability of the system is improved because conventions and coding guidelines are enforced, which directly affects the maintainability KQF. Thirdly, it may be possible to optimise the system using verification. For example, if the system uses a long and inefficient query to fetch the data, an assessor may suggest an alternative method which may improve the efficiency KQF. Verification also has a number of drawbacks. The key drawback is the need to perform the process manually, there are currently no tools which would verify the software as good as a qualified human.

6.2 Software Validation

The goal of software validation is to confirm that the product was correctly built by adhering to the specifications developed at the start of the development life cycle. It enables the organisation to prove their claim that the product matches the requirements and has the necessary degree of quality. Fundamentally, there are three types of validation which are: validation of software designs, validation of product specifications and validation of software product (Chemuturi, 2011, pp.132-135). Validation of software designs ensures that the design of the product was performed correctly and that it conforms to the user requirements and specific guidelines/standards. In software systems, validation of the software designs is achieved by creating and testing product prototypes. For complex algorithms it is possible to design simulated environments to analyse their behaviour and capture necessary feedback, allowing to avoid specific issues during the development phase.

Validation of the product specifications is a critical activity of every project because it ensures that the requirements have been captured and understood correctly. It uses techniques such as reviews to validate the specifications before the product is designed and developed. Validation of the software product validates the product's correctness by applying a range of testing techniques and tools. Fournier (2008, pp.102-103) suggests that testing should utilise artefacts in order to introduce traceability and allow the development of the correct test cases which can be used to validate the product. The aim of testing is to confirm that the chosen input correctly matches the known output (Chemuturi, 2011, p.134). Software testing is integrated into all software development lifecycles and is considered as an essential activity to validate the product's quality. A summary of the testing techniques is presented in *Table 2* to outline the methods which could be applied for validation.

TESTING NAME	SUMMARY OF THE STRATEGY
Regulation Conformance	Ensuing the system conforms to all government and domain regulations
User Manual	Assessing whether the system correlates to the user manual documentation
Black-box	Inputs are compared with the outputs without examining the internal logic
White-box	Considers the internal logic of the system by checking each branch
Unit	Individual units of the system are independently tested
Integration	All software components and combined and tested as a single system
Functional	Ensuring core and ancillary functions are working correctly
System	Checking that the system is compliant with the specified requirements
Sanity or Smoke	Basic level of testing to ensure critical functionality work correctly
Regression	Ensuring the system behaves the same way before changes were introduced
Acceptance	Checking the system for compliance with the business requirements
Load	Determining the system behaviour during peak conditions
Stress	Ensuring the system operates correctly when expected resources are absent
Performance	Evaluating the overall operating performance of the system
Usability	Checking with real users how easy the system is to use
Parallel	Several users are accessing the same function and manipulating same data
Recovery	Determining how well the system can recover from an unexpected crash
Compatibility	Ensuring the system is compatible for a specific running environment
Alpha	Real users conduct software testing at the place where system is developed
Beta	Real users conduct software testing in any convenient location
Defect	Revealing areas where the system does not conform to the specification
Positive	Providing valid data to ensure the system correctly performs functionality
Negative	Providing system with invalid input to check how it handles it
Install-Uninstall	Checking that the install and uninstall functions are correctly performed

Table 2 – Summary of the Testing Approaches

There are other testing techniques which are not covered in *Table 2*, such as: equivalence partitioning, error guessing, consistency and end-to-end testing. The table has focused on outlining the testing approaches which are relevant to the quality of the system. It is very rare that all tests currently existing are performed on the software systems. Chemuturi (2011, p.178) specifies that most organisations mainly focus on functional, integration, positive/acceptance and load testing due to time and budget constraints. Unit testing is also becoming popular due to integration with IDEs (Integrated Development Environment). It has a positive impact on integration, functional, regression and load testing because same test can be run multiple times during the development lifecycle.

The most significant benefit of validation testing is the plethora of approaches which can be adapted in order to thoroughly test the system. This enables to select the suitable techniques which will focus on testing the product against the success criteria identified at the start of the development lifecycle. However, this can be seen as a major disadvantage because with such a large number of approaches it may be difficult to select the appropriate one. In the worse scenarios, a company may want to perform all types of testing to ensure quality of the product, directly affecting the time and the cost of the project. Validation can be applied towards ensuring that the metric truly measures what the assessor intends to measure (Kan, 2002, p.71). Such strategy ensures that the metrics output true and meaningful data which can be analysed and used to improve the process and product quality. Patton (2001, pp.306-316) notes that testing can output useful statistics which can be compliment metrics in order to quantify the quality of the product. Furthermore, testing can only reveal existing bugs and not their absence (Patton, 2001, pp.41-42). The system is primary tested to confirm the absence of specific defects or defects; however, this does not mean that the product will withstand all the bugs discovered during the system's exploitation. Patton (2011, p.41) also notes that resolving all system defects, bugs and faults is not feasible due to time limit. Another variable of testing is the testers who may not be qualified, absence of system knowledge or they simply do not go into enough depth to discover all the defects, which negatively impact on the product's quality.

7 Conclusion

This report has successfully reviewed several software quality tools, techniques and theories. Fundamentally, the software quality is built-up on constant validation and verification of the product as well as the process. The report has begun by stating the aims of the project which will be used towards the end to assess the overall project. This was followed by giving a brief summary of tools and SQA work. The report then proceeded onto reviewing the four dimensions of quality and differentiating product from process quality. Several quality models were reviewed and a number of KQF were created for this project. Several suitable metrics were suggested, allowing to correctly and accurately measure the KQF to ensure the correct degree of quality is achieved. The report then focused on reviewing validation and verification techniques used to confirm specific aspects of the software development lifecycle throughout the project.

The SQA practices are certainly an important aspect of modern software development. It enables to build quality into the product by applying a range of techniques applicable to different processes. However, due to the high volume of quality practices presently available, it is hard selecting a correct approach applicable to the organisation type and the type of systems they develop. This is clearly reflected by a multitude of quality models which focus on different aspects of high-level quality factors. Furthermore, due to each product being required by different people, it is not possible to always reuse same quality factors to assess the quality of the end product. Although, quality models should be a starting point for any company wishing to improve their process and product quality. Furthermore, the KQF do not follow any prioritisation which makes it hard distinguishing which factors are more important towards the system, allowing to focus on it more. For example, maintainability KQF could be more critical than flexibility KQF. By measuring the importance of each KQF it would be possible to ensure a higher degree of quality. Additionally, the proposed metrics are not systematised in terms of applicability at different stages of the software development lifecycle. In other words, it is not clear when and how exactly the metrics should be used to measure the quality. There is also lack of guidelines in terms of metric outputs, it is not clear which boundaries represent different levels of quality. As an example, a big company may aim for 95% quality, whereas a medium company may aim for 80% quality. Both companies will deliver similar system with all critical functionality working correctly, however, the system with 95% quality is likely to cost more due to complexity of the development process. As a result, the users may not select it.

These issues are overcome by starting to follow specific models and applying the gained knowledge towards each process in order to improve the software development lifecycle. A range of metrics can be tested and suitability of each could be assessed when measuring the achieved quality of the product. Commonly, the third project done by the same development team by applying the learned knowledge is likely to be successful because team will be familiar with the quality practices and have fundamental knowledge from the previous mistakes. This enables managers to create realistic project schedules and milestones, allowing them to correctly estimate the cost of the project. The various quality certifications such as CMMI are good target wishing to improve their quality, however due to their high cost it might not be achievable for certain companies. Additionally, due to the process of gaining such a certification, the company is very unlikely to lose it even if no quality practices are applied after obtaining the certification. This is because no monitoring is done by the body. Having critically analysed the software quality theories and techniques the report will continue by reflecting on the findings and applicability towards the project.

In future, it would be highly beneficial to review other quality models such as CMMI, FURPS and SRGM to gain a comprehensive overview about the similarities and differences between each model. It would also give insights into other KQF and their metrics. This knowledge can perhaps be used to prioritise and develop a set of KQF which are applicable to the specific domains of software systems. It would also be advantageous to further research KQF and develop a systematic metric approach of each, as currently there is no concrete metric to measure a specific KQF based on a quality model. While there are several metrics for each KQF, organisations might struggle selecting the most suitable one in order to measure their process and product. Lastly, applying the theoretical knowledge of quantifying and measuring quality should be applied in practice to reveal any outstanding issues, allowing to reflect upon the suggested quality practices. This is because the software quality assurance looks fairly simple in theory, but the difficulty of applying it in practice is immense.

References

Accept360 (2018) *Product and Project Planning for a Complex World*. Available at: http://www.aisc.com/solutions/accept360/. (Accessed: 3 March 2018).

Accompa (2018) *Requirements management Software Tool – Accompa*. Available at: http://web.accompa.com/. (Accessed: 3 March 2018).

Al-Qutaish, R. (2010) 'Quality Models in Software Engineering Literature: An Analytical and Comparative Study', *Journal of American Science*, 6(3).

ApTest (2016) *Software QA Testing and Test Tool Resources*, ApTest: Software Testing Specialists. Available at: http://www.aptest.com/resources.html. (Accessed 28 February 2018).

Arisa (2009) *Software Quality ISO Standards*, Arisa. Available at: http://www.arisa.se/compendium/node6.html. (Accessed: 27 February 2018).

Atom (2018) *Atom – A hackable text editor*. Available at: https://atom.io/. (Accessed: 3 March 2018).

Axure (2018) *Design the Right Solution*. Available at: https://www.axure.com/. (Accessed: 3 March 2018).

Basecamp (2018) *Basecamp: Project Management & Team Communication Software*. Available at: https://basecamp.com/. (Accessed: 3 March 2018).

Barjaktarovic, L., Jecmenica, D. (2011) 'Six Sigma Concept', *Acta technical Corvinienis – Bulletin of Engineering*, 4(4), pp.103-107.

CaseComplete (2018) *Use Cases and Requirements Management*. Available at: http://casecomplete.com/. (Accessed: 3 March 2018).

Chemuturi, M. (2011) *Mastering Software Quality Assurance: Best Practices, Tools and Techniques for Software Developers*. Fort Lauderdale, Fla.: J. Ross Pub.

CreativePro Office (2018) *Project Management Dashboard Software to Organise Your Software*. Available at: http://mycpohq.com/. (Accessed: 3 March 2018).

Desai, D. (2010) *Six Sigma*. 1st edn. Mumbai India: Himalaya Pub. House.

Doxygen (2018) *Generate Documents from Source Code*. Available at: http://www.stack.nl/~dimitri/doxygen/. (Accessed: 3 March 2018).

DrExplain (2018) *A Software to Create Help Files, Online Help Manuals, User Guides & Documentation*. Available at: https://www.drexplain.com/. (Accessed: 3 March 2018).

Eclipse (2018) *Desktop IDEs*. Available at: https://www.eclipse.org/ide/. (Accessed: 3 March 2018).

Egyed, A., Grunbacher, P. (2004) 'Identifying Requirements Conflicts and Cooperation: How Quality Attributes and Automated Traceability Can Help', *IEEE Software*, 21(6), pp.50-58.

Farid, S., Alam, M., Akbar, A., Iqbal, M., Siddiqui, F. (2017) 'Gauging Quality of Software Products Using Metrics', *University of Engineering and Technology Taxila. Technological Journal*, 22(1), pp.121-127.

Fagan, M. (1976) 'Design and Code Inspections to Reduce Errors in Program Development', *IBM Systems Journal*, 15(3), pp.182-211.

Fournier, G. (2008) *Essential Software Testing: A Use Case Approach*. Boca Raton, Fla.: Auerbach; London: Taylor & Francis distributor.

Fossil (2018) *What is Fossil?.* Available at: https://www.fossil-scm.org/index.html/doc/trunk/www/index.wiki. (Accessed: 3 March 2018).

Galin, D. (2003) 'Software Quality Metrics – From Theory to Implementation', *Software Quality Professional*, 5(3), p.24.

Galin, D. (2004) *Software Quality Assurance: From Theory to Implementation*. Harlow: Pearson Addision Wesley.

Git (2018) *Distributed in the new Centralised*. Available at: https://git-scm.com/. (Accessed: 3 March 2018).

GoMockingBird (2018) *Website wireframes: Mockingbird*. Available at: https://gomockingbird.com/home. (Accessed: 3 March 2018).

Gorton, I. (2011) *Essential Software Architecture*. Berlin, Heidelberg: Springer – Verlag Berlin Neidelberg.

Kan, S. (2002) *Metrics and Models in Software Quality Engineering*. 2nd edn. Boston, Mass.; London: Addison-Wesley.

Khaddaj, S., Horgan, G. (2004) 'The Evaluation of Software Quality Factors in Very Large Information Systems', *Electronic Journal of Information Systems Evaluation*, 7(1), pp.43-48.

Kwak, Y., Anbari, F. (2006) 'Benefits, Obstacles and Future of Six Sigma Approach', *Technovation*, 26, pp.708-715.

Learn Six Sigma Definition (2018) *Six Sigma*, Learn Six Sigma Definition. Available at: http://leansixsigmadefinition.com/glossary/six-sigma/. (Accessed: 27 February 2018).

Manns, T., Coleman, M. (1996) *Software Quality Assurance*. 2nd edn. Basingstoke: Macmillan.

Miguel, J, Mauricio, D., Rodriguez, G. (2014) 'A Review of Software Quality Models for the Evaluation of Software Products', *International Journal of Software Engineering & Application*, 5(6).

Ming-Chang, L. (2014) 'Software Quality Factors and Software Quality Metrics to Enhance Software Quality Assurance', *British Journal of Applied Science & Technology*, 4(21), pp.3069-3095.

Miyoshi, T., Azuma, M. (1993) 'An empirical study of evaluating software development environment quality', *IEEE Transactions on Software Engineering*, 19(5), pp.425-435. DOI: 10.1109/32.232010.

Mogups (2018) *Online Mockup, Wireframe & UI Prototyping Tool*. Available at: https://moqups.com/. (Accessed: 3 March 2018).

Nanz, S., Furia, C. (2015) 'A Comparative Study of Programming Languages in Rosetta Code', *Proceedings – International Conference on Software Engineering*, vol.1, pp.778-788.

NetBeans (2018) *NetBeans IDE – The Smarter and Faster Way to Code*. Available at: https://netbeans.org/features/index.html. (Accessed: 3 March 2018).

Patton, R. (2001) *Software Testing*. Indianapolis, Ind.: Sams.

Schulmeyer, G. (2007) *Handbook of Software Quality Assurance*. 4th edn. Norwood: Artech House.

Scoro (2018) *Scoro: The Most Comprehensive Business Management Software*. Available at: https://www.scoro.com/. (Accessed: 3 March 2018).

Software Quality Management (2016) Available at:
http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management. (Accessed: 23 February 2018).

Sommerville, I. (2010) *Software Engineering*. 9th edn. Harlow: Addison-Wesley.

StarUML (2018) *Star UML 2*. Available at: http://staruml.io/. (Accessed: 3 March 2018).

Tomar, A., Thakare, V. (2016) 'The Selection of Programming Language to Reduce Defect and Increase Quality', *International Journal of Scientific & Engineering Research*, 7(2), pp.148-150.

Visual Studio (2018) *Visual Studio IDE*. Available at: https://www.visualstudio.com/vs/. (Accessed: 3 March 2018).

Zhu, H. (2005) *Software Design Methodology: From Principles to Architectural Styles*. Elsevier Science.