A framework for software quality assurance

SPJ ABO'O ZO'O

orcid.org/0000-0003-3021-8986

Dissertation accepted in fulfilment of the requirements for the degree *Master of Science in Engineering Sciences with Computer and Electronic Engineering* at the North West University

Supervisor: Dr J van Rensburg

Graduation: July 2021

Student number: 30645948

ABSTRACT

Title: A framework for software quality assurance

Author: S P J Abo'o Zo'o

Supervisor: Dr J van Rensburg

Keywords: Software development, quality assurance, quality control, code, development framework, quality standards, automation, continuous integration

The current era is highly impacted by new technologies, creating a technological race, including new features and updates daily, which produces a significantly high amount of code that is generated. However, as an organisation grows, it becomes harder to maintain software quality standards, leading to a bigger emphasis on creating new features instead of the quality of software throughout the development processes, which allows for defects to slip through.

The use of existing tools and techniques is highly advisable to ensure and enforce the quality of software products. The main practice emphasised in this study is software quality assurance, which goes hand in hand with software control. The discipline of software quality assurance is the procedure of verifying that a piece of code meets the processes, standards, and specification requirements. Conversely, software quality control focuses on the outcome of the solution, ensuring the proper functioning of a product, whilst ensuring that its quality will indeed mitigate defects.

While implementing those practices, several tools and technologies must be used to guarantee overall efficiency. The focus is on Continuous Integration, which is a practice of automating the code merging process while implementing a series of verification steps. This safeguards the program's overall quality, as well as the outcome of the solution. However, moving from a manual to an automated process requires high monitoring and a defined and well organised sequence of events.

The goal of this study is to design and implement a software quality assurance framework; one that can be generic and easily implemented, incorporating the concepts central to this study, and according to a company's standards. The automated framework results in a faster, less error-prone and more robust software development process. This will ensure a product of quality, thus increasing the value of the software as a company's asset.

Based on the study's need, the proposed framework focuses on the business needs in a generic manner, using available tools and techniques. It is adaptable to any given piece of code or feature. It comprises a planning stage, which focuses on the requirements and standards of quality. It also has an implementation stage where the coding is performed, tested, verified, implemented, and maintained.

Some quantifiable results obtained from implementing the framework showed an increase in potential errors detected with a defect reduction, as well as a decrease in the verification time. The former is the number of errors that can have a negative impact on the outcome, which are detected through each iteration, while the latter relates to the time that is physically needed for a human to perform verification steps manually.

By enforcing and improving implementation of the framework, the defects obtained in the form of error and critical errors decreased by 40% on average. The number of warnings increased by 40%, resulting in more potential defects being detected. Also, the verification time decreased by an overall 25%. The results obtained confirm that the proposed framework assists in mitigating and controlling system defects. Quality is, therefore, assured.

ACKNOWLEDGEMENTS

I would like to extend my immense gratitude to:

- Almighty God, for being my ultimate provider. I thank Him for His abundant favour and infinite Grace upon me as He made this opportunity possible. I give Him all the glory and praise forever,

- My parents and pillars of strength Prosper Zo'o Minto'o and Sylvie Hermine Andeme, and my beautiful family. Thank you for your constant love and support.

- Professor E. H. Matthews, the CRCED Pretoria, Enermanage (Pty) Ltd and its sister organisations, Dr J. van Rensburg and Dr J. N. du Plessis.
  Thank you for providing me with the required funding, resources, leadership, and guidance, which contributed to the completion of this study,

- My dearest friends and colleagues, thank you for the endless encouragements. And a special thanks to Pieter Engelbrecht for the valuable insights.

TABLE OF CONTENTS

ABSTRACT I

ACKNOWLEDGEMENTS III

LIST OF FIGURES VI

LIST OF TABLES VII

LIST OF ABBREVIATIONS VIII

LIST OF TERMS IX

1 INTRODUCTION 1

1.1 INTRODUCTION 1
1.2 PROBLEM STATEMENT 2
1.3 NEED FOR THE STUDY 3
1.4 OVERVIEW OF DISSERTATION 4

2 LITERATURE REVIEW 6

2.1 OVERVIEW OF QUALITY ASSURANCE IN SOFTWARE DEVELOPMENT 6
2.2 SOFTWARE QUALITY CONTROL 13
2.3 AUTOMATION OF THE SOFTWARE DEVELOPMENT PROCESS 16
2.4 SOFTWARE TESTING 21
2.5 PREVIOUS STUDIES 24
2.6 SUMMARY 28

3 DEVELOPMENT OF THE SOLUTION 30

3.1 INTRODUCTION 30
3.2 DEVELOPMENT OF THE FRAMEWORK 30
3.3 UNDERSTANDING THE REQUIREMENTS 32
3.4 INVESTIGATING QUALITY STANDARDS 33

3.5 DEVELOPMENT OF QUALITY CHECKS 35
3.6 AUTOMATION 36
3.7 DESIGNING THE WORKFLOW PLAN 41
3.8 PIPELINE IMPLEMENTATION 43
3.9 VERIFICATION 48
3.10 SUMMARY 49

4 RESULTS AND DISCUSSION 52

4.1 INTRODUCTION 52
4.2 PROPOSED FRAMEWORK 53
4.3 IMPLEMENTATION OF PIPELINE 58
4.4 ADDITIONAL BENEFIT 65
4.5 OUTCOME 67
4.6 SUMMARY 69

5 CONCLUSION AND RECOMMENDATIONS 72

5.1 CONCLUSION 72
5.2 LIMITATIONS OF THE STUDY 74
5.3 RECOMMENDATIONS FOR FUTURE WORK 75

REFERENCES 76

APPENDIX A – EVENT LOGS 83

APPENDIX B – CODE COVERAGE 88

APPENDIX C – CODING ACTIVITY 90

LIST OF FIGURES

Figure 2-1: Software quality assurance processes [34] 9
Figure 2-2: Implementing software quality in agile development [37] 10
Figure 2-3: Software quality metrics [44] 15
Figure 2-4: Overview of Continuous Integration 19
Figure 2-5: Summary of software testing [67] 23
Figure 3-1: Development of the framework 31
Figure 3-2: Summary of DevOps using GitHub, Docker, and Azure 40
Figure 3-3: Sample workflow plan 42
Figure 3-4: Pipeline implementation 45
Figure 3-5: Sample task 46
Figure 3-6: Summary of pull request 48
Figure 4-1: Composition of the framework 52
Figure 4-2: Legend of the framework 53
Figure 4-3: Proposed framework 55
Figure 4-4: Error reduction over time 60
Figure 4-5: Critical error reduction over time 61
Figure 4-6: Defect detection over time 63
Figure 4-7: Verification time 65
Figure 4-8: January test coverage summary 66
Figure 4-9: November test coverage summary 66

LIST OF TABLES

TABLE 2-1: SUMMARY OF SOFTWARE QUALITY STANDARDS CHARACTERISTICS 12
TABLE 2-2: STATE OF THE ART MATRIX INDICATING GAPS IN PAST STUDIES 27
TABLE 3-1: CODING STANDARDS 34
TABLE 4-1: ERROR REDUCTION OVER TIME 60
TABLE 4-2: CRITICAL ERROR REDUCTION OVER TIME 61
TABLE 4-3: DEFECT DETECTION OVER TIME 62
TABLE 4-4: VERIFICATION TIME 64
TABLE 4-5: CODING ACTIVITY FROM JANUARY TO OCTOBER 2020 68

LIST OF ABBREVIATIONS

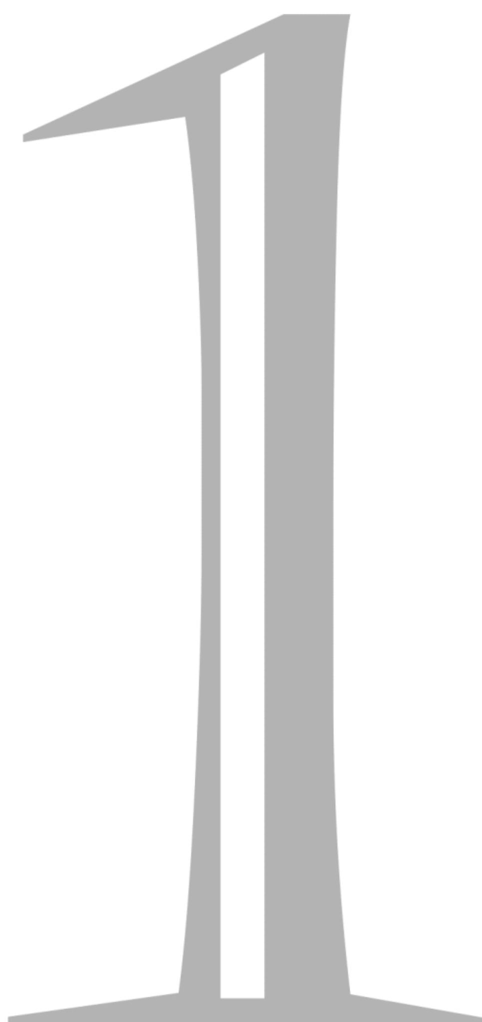
CI	Continuous Integration
IDE	Integrated Development Environment
ISO	International Organisation for Standardisation
IT	Information Technology
QA	Quality Assurance
QC	Quality Control
SDLC	Software Development Lifecycle
SQA	Software Quality Assurance
SQC	Software Quality Control

LIST OF TERMS

Term	Definition
Assurance	The aptitude to provide a guarantee in case of risks.
Code	Computer code is a language that allows a human to program software.
Framework	A framework is a combination of ideas, rules, or beliefs, which are used to resolve problems or to choose a plan of action.
Quality	The ability of a product to satisfy the needs expressed or implied by consumers.
Software	A set of programs and procedures necessary for the operation of a computer system.

CHAPTER 1

Introduction



1 INTRODUCTION

1.1 INTRODUCTION

The current context of the software market emphasises cost, timing and functionality [1]. Software quality and quality assurance are often relegated to the background. Most developers do not understand that poor software quality can have negative impacts such as high costs and schedule delays [2]. For many organisations quality verification only occurs at the time of testing, while a significant part of the development budget is then devoted to correcting the induced errors; often projects spend a large portion of their budget on recovery costs [3].

The word "quality" can be defined as the property, attributes or state of things or people, which make it possible to distinguish them from others and to determine their nature. Quality refers to the measurement of the value of certain key attributes that are identified from the outset as carriers of quality [4].

A product is said to be of quality when it meets the initial specifications with regards to the scope, time and cost requirements [5]. A requirement is a condition or specification that must be met or attribute that must be possessed. A requirement can be functional, which describes what the system should so; or non-function, which describes how it should be done [6]. Software requirements are gathered through several techniques, which involve interactions with the potential users. The most common techniques include interviews, questionnaires, brainstorming, storyboard, use cases and prototyping [7].

Ensuring quality means ensuring that all the requirements gathered are met. In software development, not all attributes are measurable, since they have unquantifiable logical values at times. The ISO defines standards for quality in software development and engineering [8].

Quality is increasingly perceived to be a critical influencing parameter in business and an important driver of customer satisfaction [9]. Its absence can accrue consequences, which include heavy financial losses, dissatisfaction amongst users, and damage to the environment that can even cause deaths as an ultimate and serious consequence [10]. It is, therefore, highly advisable for organisations to prioritise the quality of their software.

However, quality is often neglected when developing applications owing to the technological race. Instead, more emphasis is placed on coding and launching as opposed to efforts to ensure

controlling and testing the quality of the code. Where code quality is neglected, software defects are on the rise; therefore, more resources are allocated to maintenance, which is a waste [11].

The main objective of each software development process should be to ensure a product of quality, which is ready to meet the needs of end-users, customer requirements and business objectives [12]. Quality is an indispensable attribute of every software solution. To guarantee this quality and to provide an accurate assessment, numerous methodologies and techniques can be applied [2].

Quality assurance, quality control and quality planning form the essential activities involved in managing software quality [13]. Quality assurance and control, which constitute the focus of this study, are among these methodologies. Although the concepts are interdependent, they are neither substitutable nor interchangeable.

Organisations need to implement a quality regulator plan that encompasses quality assurance and control, as well as automated software testing [13]. Following an automated approach to avoid the disadvantages and time concerns of a manual one, a generic and automated framework could help a company reap the benefits of quality software, as explained before [14].

1.2 PROBLEM STATEMENT

Quality is a vital characteristic of a software product and cannot be neglected. The risks associated with poor software quality are costly on a financial and technical level and ultimately affect the software's quality [15].

Assuring and controlling quality are essential for any company that creates and designs a product or service intended for use by third parties [16]. In this case, the product is the software, and it will require improving the quality requirements. Regarding software design, some players in the development chain want quality to start only once development has been completed [17]. Quality must already be a non-negotiable criterion from the first hours of reflection, as well as before, during and after development [18].

1.3 NEED FOR THE STUDY

To guarantee the software's quality, the discipline called Software Quality Assurance (SQA) makes it possible to define all the quality requirements upstream, and to agree to their fulfilment at each stage of development until delivery of the solution [19]. It is the set of activities allowing for assurance that processes are established and are improved continuously to provide products that meet specifications, and tailored for the intended use [20]. Quality assurance cannot be implemented without Software Quality Control (SQC), as this ensures that quality is preserved throughout the product's lifecycle.

Managing software quality should be a universal activity to ensure that the product software's quality meets customers' needs. Hence, there are key metrics, which must be checked at every step of the software development process [21]. To optimise all benefits of the SQA process, it is highly advised to implement automation via the use of CI rather than opting for a manual process [22]. The reason for this is that it can be rushed because it is time consuming, and, consequently, more prone to errors.

Therefore, the need for the study is based on the need for a well-defined generic and automated SQA framework that addresses SQA and SQC based on quality standards, whilst following an appropriate process with precise steps [12]. A SQA plan should be followed to ease the process and to implement better ways of controlling and testing the quality of the code [18]. A proper generic and automated SQA framework minimises the number of defects [3].

The creation of a software product does not only comprise of coding. The process also consists of project management, specifications of functionalities, constraints, quality factors, and interface design. The software development process is functional and architectural. It includes algorithms, a detailed design, source code, executables, and tests [23]. The importance of testing is seen in case of problems. However, it is not advisable to wait for problems to occur to take the issue seriously.

1.3.1 OBJECTIVES

The study's main objective is to develop and implement a framework for SQA, which:

- Follows the principles of SQA;
- Is automated; and

- Is generic.

1.4 Overview of Dissertation

Chapter 1: Introduction

This chapter introduced the study and identified the research problem statement as well as the study objectives.

Chapter 2: Literature Review

This chapter provides an overview of the concept of quality in terms of software, and defined SQA and SQC. It also provides an investigation of previous studies that relate to the topic and depicted the need for the study.

Chapter 3: Development of Software Quality Assurance Framework

This chapter provides an explanation of the development of the solution, which is the SQA Framework. It details the steps taken to reach the solution, while defining the methodology and its implementation.

Chapter 4: Results and Discussion

This chapter provides the obtained results from implementation of the solution, mostly qualitatively, as most of the expected results cannot be quantified. However, it provides a few results from the measurable quality metrics.

Chapter 5: Conclusion

This chapter concludes the study based on the acquired results. It also gives recommendations for future studies in respect of the shortcomings that this study revealed, as well as potential recommendations for improvements.

CHAPTER 2

Literature Review

2 LITERATURE REVIEW

2.1 OVERVIEW OF QUALITY ASSURANCE IN SOFTWARE DEVELOPMENT

Software can be considered to be "quality" when it meets or exceeds customer expectations and satisfies them [24]. To further clarify this, the term "quality" means more than utility, robustness, efficiency, reliability, maintainability, portability, and reusability. It includes flexibility, orchestration, the automation process, and integration with external software [12]. Quality assurance (QA) lowers costs and eliminates waste, potentially ensuring large-scale profits for the customer. In addition, software should be well maintained and improved over time to continue to add value to customers [3].

QA focuses on quality assessment processes, including planning, organisation, and implementation. The main objective of QA is to ensure the optimised and efficient means of safeguarding the predicted quality of a product [20]. QA helps not to detect problems, but to prevent them [3]. QA should be a proactive means of verifying the software quality. Quality assurance is embedded into the SDLC and requires involvement of the entire project team [19].

The quality assurance process includes planning, preparation and execution of tests, which are specific to the industry, as well as managing test information [25]. Specifying and establishing software development requirements and quality verification criteria are useful to reach an acceptable level of quality in the software, whilst enhancing productivity within the project team. It is unacceptable to add quality to software when the development process ends. It is necessary to define the specifications and to choose the metrics that should be evaluated from the start of development, and then to follow these throughout the process [12].

SQA is the set of activities, which makes it possible to ensure that the processes are established and can be continuously improved without having to start all over again [19]. This leads to the design of a product, which meets the functional specifications of the client and, which technically provides certain assurances for operations [26]. All the quality requirements that necessitate verification must be documented in the quality assurance plan and follow rigorous monitoring [12].

This plan also includes a test plan that measures other performance indicators that could not be measured in previous phases of development. The plan also assists in assessing quality costs,

which can sometimes be difficult for small projects as providing such details can be too demanding. But for complex and especially sensitive projects, involving critical or real time actions, quality should be present [3].

In an agile development environment, requirements can change late in the development process, and a system must be established to accommodate such changes without altering the software quality [27]. To better understand and to distinguish all the comprised parts, it will be necessary to detail the different elements that accompany the process of ensuring the software quality from the perspective of the three main stakeholder. They are considered here to be the main stakeholders because they may be the mostly affected [27].

- **Quality from the perspective of the user or client**:

The quality of the software this stakeholder is assessed based on end users' interaction with the software [18]. For the customer, the quality of a system depends on receiving a response that satisfies their aesthetics, usability, reliability, ergonomics, adaptability and functionality requirements [24].

In other words, is the software doing what it is supposed to do in a quick, easy, and reasonable manner? There are several similar questions, which may seem trivial, but which are significant during delivery. A product can be rejected merely because of the external appearance, and even if all the features work [1]. It is important to have answers for all these requirements to ensure that on the clients and users' side, the software meets quality, as they ultimately assess the system's quality metrics based on whether or not their needs are met [24].

- **Quality from the perspective of the software development team**:

The software development team includes the architect, analyst, software engineer or developer. According to this stakeholder, quality has another meaning and is not measured in the same way. It assumes a more professional turn and is not measurable by the client [28]. Software quality is generally perceived here as the number of faults, and the ability to make the system evolve and perform tests on the system [29]. Recent technological evolutions, the nature of the design, modelling technique, architecture and database choice, compliance requirements, development language, as well as the deployment platform should be considered [30].

Attention should be given to the simplicity and modularity of the system, the documentation produced throughout the process, and justification for the choices [4]. The software engineer

or developer must carefully choose and record the processes, as well as tools and techniques to monitor and control the quality of the software during design [3].

- **Quality from the perspective of the quality assurance team:**

In the case of a medium to small organisation, this stakeholder is represented by the development team. The quality engineers along with all the other members of this team have an attribute list that must be checked and validated to ensure the quality of any software [31]. Certain standards must be met. However, they must participate in the entire development process and must consider all points of view to test the quality.

They consider all the requirements and highlight the attributes that they have been able to identify and that can contribute to the software quality [32]. These attributes will be added to their initial list for monitoring. As the software development progresses, they must ensure that quality objectives are met. In applying standards, they must be strict [4].

If it is difficult to control all the quality attributes of the software, it is imperative to ensure that qualitative processes are established, as the success of these will contribute to achieving quality objectives, especially on the part of the user or client. An optimised and less complex process will contribute to quality assurance [20].

The main questions that any quality assurance team should address revolve around the defined quality factors, pre-established standards and the implementation of quality during the development stage on a technical level [2]. Any flaw in this process can impact the quality of the result. Key elements of quality assurance that must also be considered are [33]:

- standards;
- examinations and verifications;
- tests;
- collection and analysis of errors or failures;
- change management;
- customer management; and
- security and risk management.

2.1.1 PROCESSES INVOLVED IN SOFTWARE QUALITY ASSURANCE AND CONTROL

SQA should be enforced in all stages of the product development life cycle, as shown in Figure 2-1. First, during the initial phases of the development process, it is paramount to identify the quality requirements during the research and analysis stages, as it will provide a baseline to follow for the other steps [3].

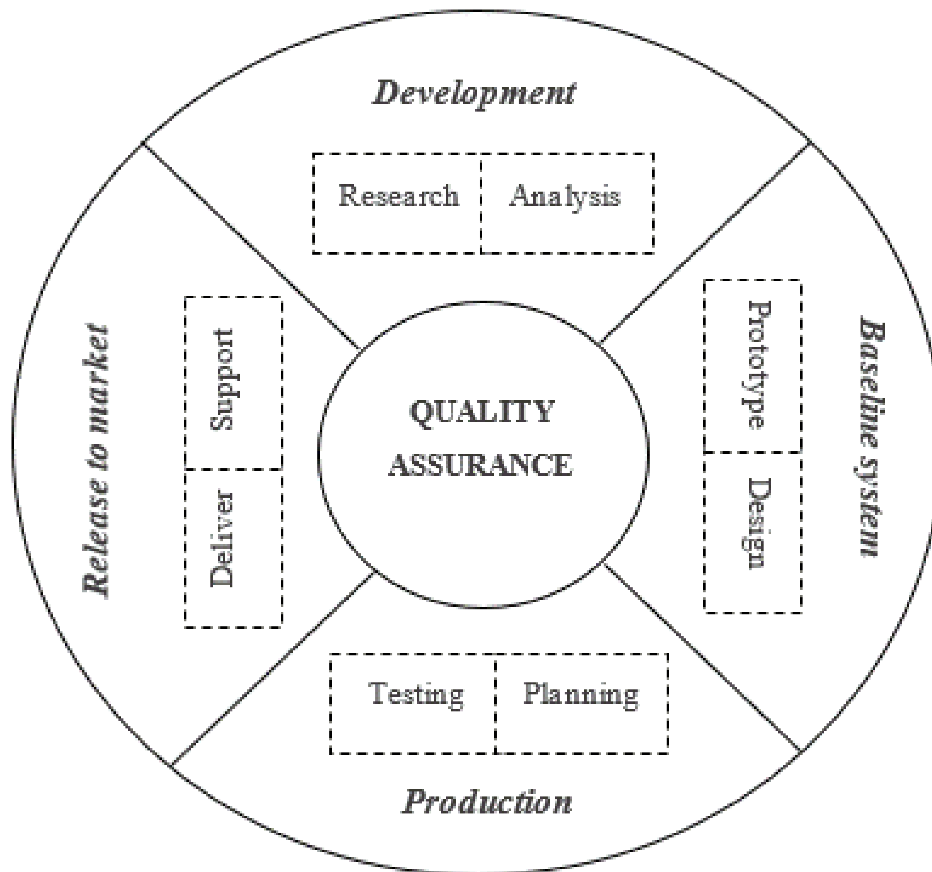


Figure 2-1: Software quality assurance processes [34]

This baseline will be used as a starting point in the design of the prototype. Thereafter, the design specifications are tested during the production phase and, if all goes well, the product is ready to be released. It is important to ensure that the factors of cost, time, and scope requirements of the products have been considered [11].

The processes begin with research to identify the quality standards in a generic manner. The needs analysis obtains the client's requirements and models them to move on to design [35]. This makes it possible to agree with the client in terms of these needs and to collect information

on the description of the processes from the various actors. Analysis determines, which quality standards apply to the product at hand during the development process [4].

Then, during system design, prototyping helps to ensure that the quality standards are properly implemented and respected. After prototyping, testing is paramount. Software testing enables the developers to pinpoint potential defects with different pieces of code and, most importantly, to ensure that the output, based on the given input, is as expected [36]. In other words, this ensures that the product does what it is supposed to do. Lastly, when releasing the product, support must be provided for errors and bugs not accounted for, as well as maintenance of the software support [3].

As mentioned above, providing assurance to software quality cannot be an isolated activity. It must be part of the development process and be augmented at each stage of the product development [4]. For that reason, the processes of SQA are closely related to the phases of the development process. Figure 2-2 outlines the phases of a basic agile development process.

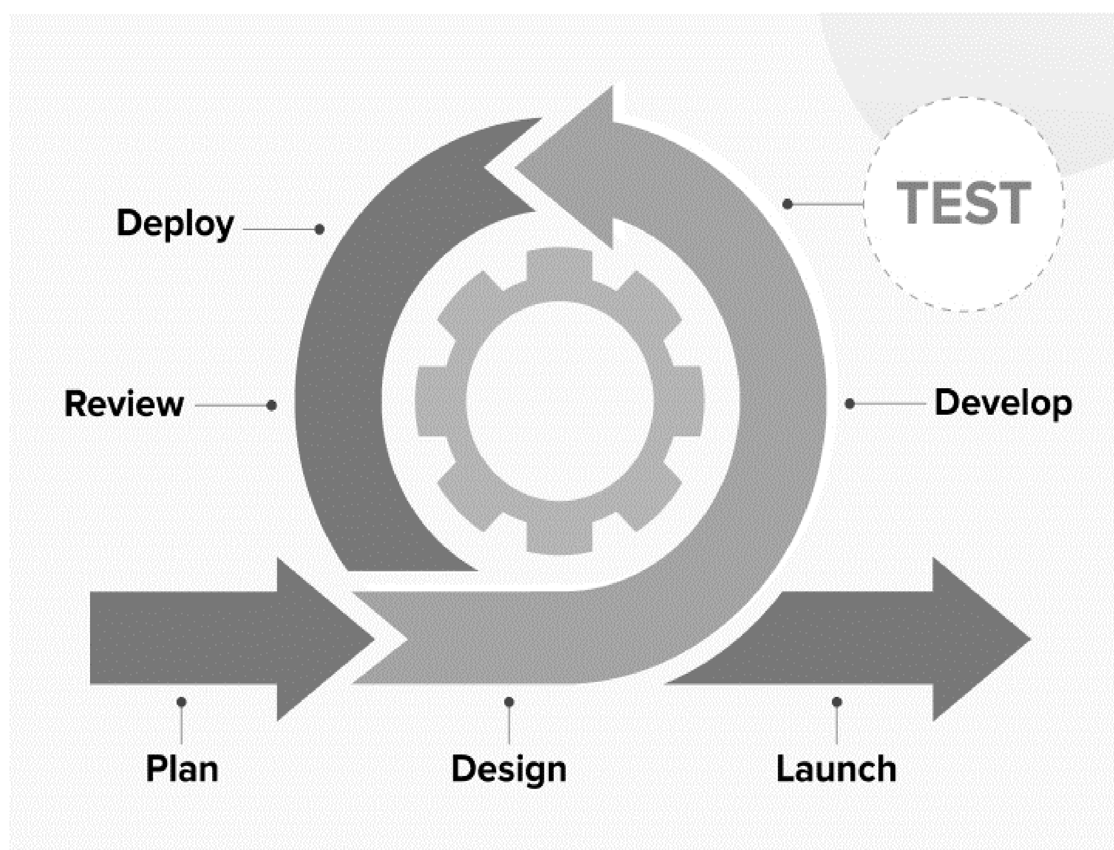


Figure 2-2: Implementing software quality in agile development [37]

The agile development process begins with planning followed by the design and development phases. This links back to the development process of SQA where a research and analysis is necessary for planning in order to be able to proceed with the prototype design, which form part of the baseline process in SQA.

In agile development, the potential solution needs to be tested fully, which is similar to the production process of SQA, in which a series of tests are executed and planning for release is performed. In agile development, the last activities include the deploy, review and launching phase. Linking back to the SQA processes, similar activities are found during the release to market process. This process includes delivering or launching the product as well as providing support, or review.

2.1.2 Software quality standards

Like any other industry, the software industry has established bodies that regulate standards. Standards determine the baselines in terms of what is acceptable from the finished product [17]. Considering the rapid growth of the discipline, it is a dynamic environment and, besides the predefined sets of standards, each individual organisation may derive their own sets of standards based on their specific requirements [38].

The software quality standards that are considered to be universal in software development, are set by the ISO 9000 group [8][26]. Table 2-1 outlines a list of the main standards characteristics [13][24][26]. The quality standards are closely related to the software requirements, both functional and non-functional, as both requirements and quality standards are characteristics that the system must possess. In other words, these characteristics are expected from software to satisfy the user experience, and determine whether the solution does what it is expected to do [39].

Table 2-1: Summary of software quality standards characteristics

Characteristics	Definition
Utility	Ability to satisfy a need or create favourable conditions for this satisfaction.
Functionality	The software functions satisfy expressed or implied needs.
Robustness	Solidly built and able to withstand extreme stresses and prolonged use.
Efficiency	Ability to produce maximum results with minimum effort.
Reliability	Ability to operate when needed without failure and for a specified period.
Maintainability	Ease of detecting and correcting errors.
Portability	Ability to interact with other systems or other platforms.
Re-usability	Ability to produce coherent, self-sufficient software modules that can be derived or re-used at any time without being recreated from scratch.
Testability	Ease of running a series of manual or automated tests.
Flexibility	Ease of upgrading or adapting the software.
Integrity/Security	Ability to withstand intentional attacks or the blunders of the user.

2.1.3 RISKS ASSOCIATED WITH POOR SOFTWARE QUALITY

A risk is any event that threatens the well-functioning of software, and the success or timely completion of a project [4]. In software development a risk can be tangible or intangible and can be grouped in different classifications [40]. This study focuses on the tangible risks. A risk can be the cause of a defect in the software and ultimately software failure. These defects can assume the form of a bug, glitch, fault, the application crashing or even a security vulnerability. This can be costly, mostly financially, for an organisation in terms of resources [41].

It is advisable to establish measures to identify risks and then to plan accordingly. This includes a risk index, a risk analysis and a risk assessment [40]. Preferably, the chosen method should be performed before starting a project, as well as prevention measures to avoid unfortunate circumstances [35]. It is also important to involve a series of tests, preferably automated, as a good software testing procedure helps to identify potential problems, whilst ensuring that the system does not crash if encountered [42]. Effective risk management will ensure defect prevention.

2.2 SOFTWARE QUALITY CONTROL

SQC emphasises the quality of the product [9] by ensuring that confirms that all standards are met and satisfy all stakeholders mentioned previously. Its main objective is to guarantee correct implementation of the established processes at the quality assurance stage [43]. Quality control activities are required to find software problems and to ensure that the issues do not reproduce continuously [9]. SQC is a reactive approach, which makes it possible to confirm that the results that are obtained correspond with those that were expected [13].

It is important to control quality during the different phases of the development process. SQC allows the software development process to be monitored to ensure that appropriate quality standards and SQA are respected [13][3]. The outcomes of the development process are compared to the standards of the project, as defined in the quality control process. The quality of the deliverables of the project can be verified by consistent software quality checks [33].

The process of reviewing the software's quality is done by a team that reviews the product and processes to verify that the predefined standards of the project were respected and upheld, and that the product and documentation comply [38]. SQC also concerns ensuring that the

requirements and characteristics of all the stakeholders in software quality are met, according to the definition that they received in the early stages [35].

Quality control includes examining the product in accordance with predetermined requirements [9]. Compared to quality assurance, quality control takes longer. Quality control activities can only be conducted after the QA procedure as they ensure that the QA procedures are properly implemented [13]. After the design and the development come SQA activities, followed by QC activities.

A software test life cycle includes QC activities. The effectiveness of QC depends solely on a test team, or in the case of a small organisation, the development team [12]. SQC is a continuous process, which occurs throughout the life cycle of the product [43].

SQC focuses on evaluating the software with the aim of guaranteeing that the metrics of software quality are met [3]. A software metric is a measure that relates to all attributes of software or process quality. The categories of software metrics are outlined in Figure 2-3, as explained by [44]. Dynamic metrics are metrics that relate to the behaviour of the program during its execution, such as the response time.

Conversely, static metrics such as code size are based on estimations made from the system's architecture. This can be the program, design, or the documentation [35]. If an external quality attribute of the software is not directly measurable, the external one is linked to a certain internal one; assuming that a relation exists between the attributes [44].

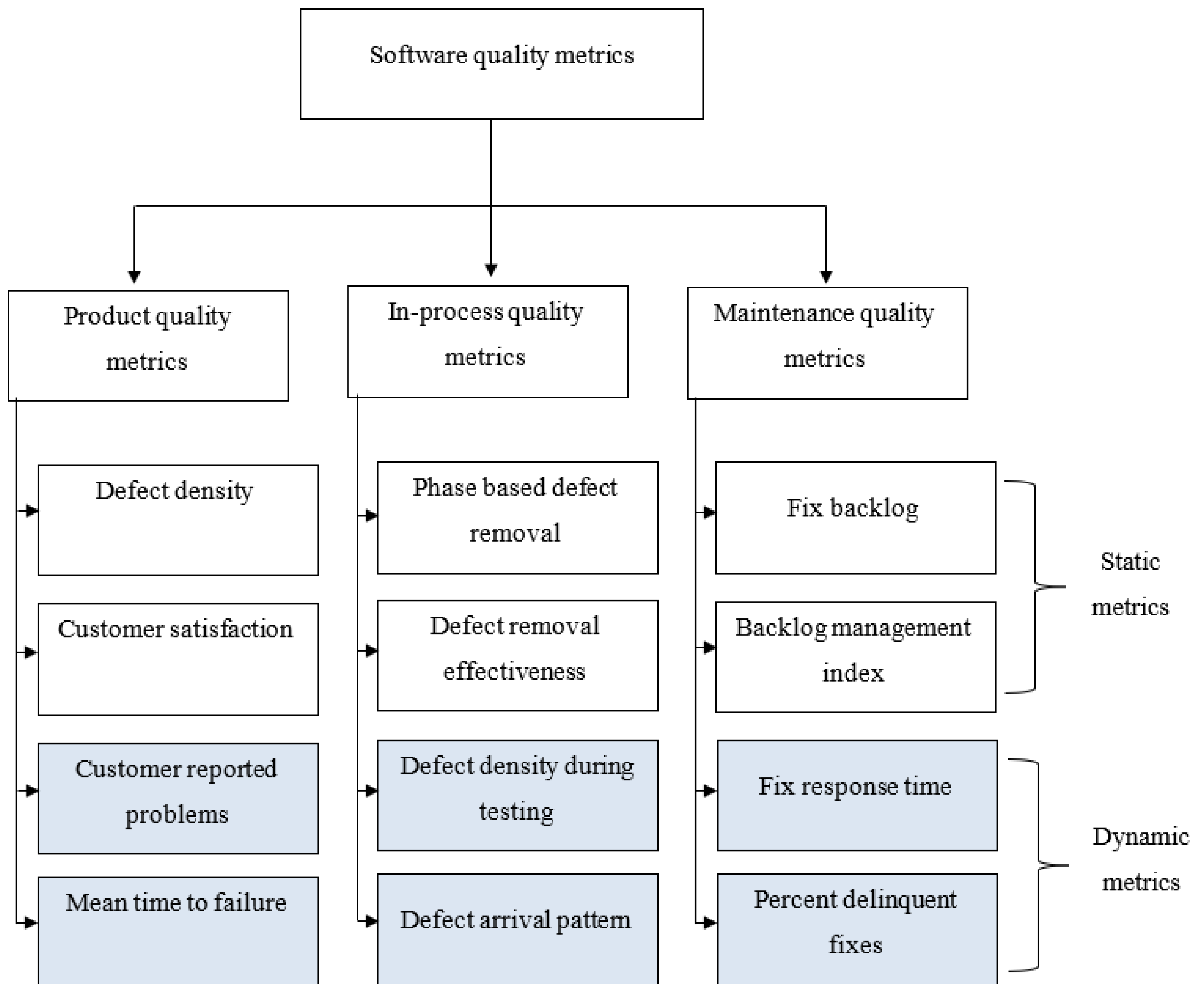


Figure 2-3: Software quality metrics [44]

2.3 AUTOMATION OF THE SOFTWARE DEVELOPMENT PROCESS

In software development, automation is the use of software and systems to perform automated processes and tasks to replace the manual and time-consuming work of people [32]. With the advent of virtualised networks and cloud services, automation has become an essential strategy to help software development teams deliver services that are faster, more consistent, and more secure [45].

This approach, which involves replacing manual tasks with processes performed by software, improves agility and increases productivity. In the software development process, automation refers to all tools and practices that are used to automate tasks, which humans perform normally [46]. It aims to do more, better, and faster with minimal or even no human intervention. To implement this means that repeatable instructions and processes are created by using scripts and task schedulers.

This approach offers several benefits for organisations. First, it limits the time spent on manual and repetitive tasks [46], reducing labour requirements, as well as operating costs. Teams are more productive and can devote more time to important tasks [47]. Automation also offers great flexibility. It decreases the risk of error, which is high in repetitive tasks, and prevents unauthorised changes. Automation is a real necessity for any organisation that seeks to improve the quality of its services and the efficiency of work teams.

In software development, practices that use automation are becoming more popular owing to their benefits. Such practices involve DevOps, as well as Continuous Integration, which are discussed further below.

2.3.1 DEVOPS

DevOps is a movement or approach that emphasises close collaboration between development (Dev) and operations (Ops) teams for any IT solution [48]. DevOps makes use of automation tools to reduce human involvement when performing software development practices [46]. DevOps combines two trades: the software developer on one hand, and the IT administrator on the other.

The aim with this merger is to improve the quality of work and the relationship between these two teams. Each has their own vision to achieve customer satisfaction - delivering new code

on demand and maintaining service availability [32]. DevOps converges with Quality Assurance as both practices encourage the implementation of software testing and collaboration in order to ensure quality throughout the development process [49].

DevOps offers a framework designed to boost and improve application development, whilst accelerating the delivery of new features, software updates or products [50]. From development to operating applications, including their deployment, its skills cover a wide spectrum and require a certain technicality and versatility [49].

This closer relationship between Dev and Ops is reflected in every phase of the DevOps lifecycle: initial software planning, coding, development, testing, release, deployment, operations, and ongoing monitoring [51]. Those steps are closely related to the SQA processes outlined in section 2.1.1, namely development where the project planning occurs, baseline system, where the solution is designed, production where the testing strategy is planned and release to market, where the solution is delivered and support is continuously provided.

DevOps generates customer feedback consistently, which reinforces the potential to improve during development, testing and deployment [46]. An example is the accelerated and permanent publication of changes or additions to features. The goals of DevOps are organised around four categories, namely culture, automation, measurement and sharing [51].

In each of these areas DevOps tools improve the streamlining and collaboration of development and operations workflows by automating the time-consuming, manual or static tasks of the integration, development, testing, deployment or monitoring phases [49][50]. By promoting communication and collaboration amongst teams that are responsible for development and IT operations, DevOps aims to optimise customer satisfaction and to deliver value-added solutions faster [32].

DevOps is also designed to stimulate innovation with a view to continuous process improvement [52]. DevOps practices accelerate, optimise and secure the business value of enterprises, for example, through more frequent release or faster delivery of versions, features or updates of products, whilst ensuring quality levels and appropriate safety devices [32]. Another goal is to improve the time that it takes to detect, resolve bugs or other problems and re-release a version [53].

An important concept of DevOps is containerisation. A container wraps the software application in an imaginary box along with all the required operating resources [54]. The resources include the actual code of the application with associated libraries, system tools, an operating system and runtime. Containers are constructed from a digital image [55].

The created image thus embeds the code and its dependencies, but it is the operating system of the host, which is used. Conversely, when using virtualisation, a complete emulation of the operating system is done, which is then used to operate the various processes. Containerisation, therefore, makes it possible to use fewer resources [55].

Containers are portable, lightweight, and afford developers to build, deploy, and run distributed applications efficiently. In addition, containerisation facilitates the database interaction. The containers also allow packaging of an application so that it can be moved easily, maximising the effortlessness [54].

2.3.2 CONTINUOUS INTEGRATION

Software development is a journey of pitfalls. However, certain practices and methodologies make the experience less tedious. This is the case of Continuous Integration (CI). CI is a DevOps practice, which aims to avoid integration issues that can quickly become negative for developers [56]. CI comprises continuously integrating changes made to the code of a software project [46]. It allows a team of developers to monitor new additions to the code continuously to immediately detect and correct any errors in smaller batches by testing them [29].

CI can be perceived as a control tower with several operators in charge of successively launching the different tools, allowing for validation of the overall functioning of the application [57]. The CI platform allows the launching of all the other tools as quickly as possible to validate a build before starting a deployment or merger of a branch, for example [58].

Generally, all members of a development team join in their progress preferably once a day. Therefore, several integrations are performed daily. An automated build checks and tests each integration to detect potential merging errors as quickly as possible [32]. This approach generally reduces the number of integration issues and allows a team to develop their software faster [57].

Automating processes involved in the building, testing and deployment simplifies development immensely. Incorporating changes more frequently also detects errors quicker to avoid unpleasant surprises that may arise from an error made several months ago [56]. Figure 2-4 shows an example of such a process.

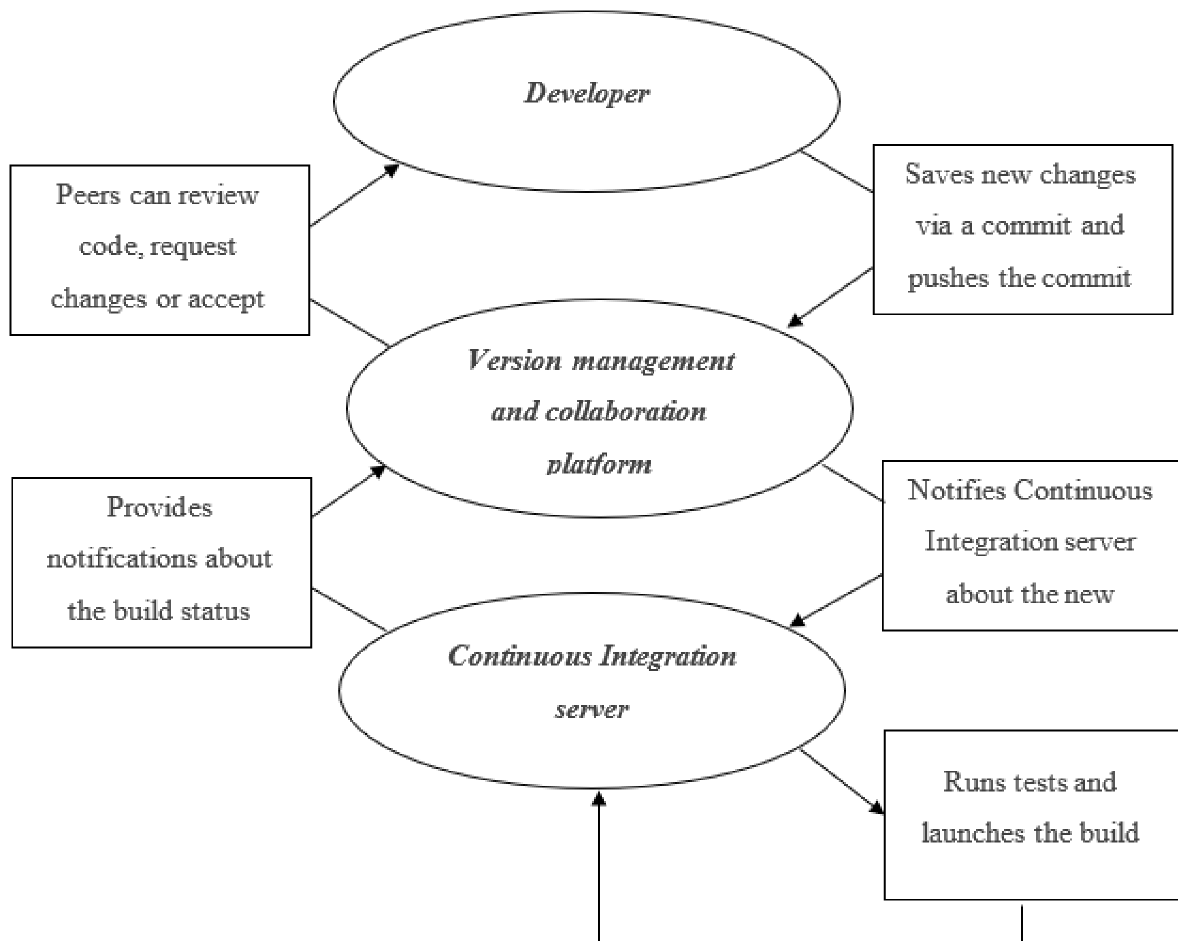


Figure 2-4: Overview of Continuous Integration

CI has several advantages. First, it reduces risks associated with integration. In fact, most of the time several people work on separate tasks as part of one project [52]. However, the greater the number of people that work on the same project, the riskier the integration process. If a problem occurs, debugging and solving it can be difficult and might require changes to the code [46]. By incorporating changes daily or even more frequently, the risk is minimised, as it is easier to control and trace back to a more recent and smaller addition [22].

Another advantage of CI is improvement of the code's quality, which improves the quality of the software consequently. Developers no longer need to worry about problems that relate to the integration and can focus more on the product's functionality [22]. In addition, in the case

of team members making an error, which "breaks" the build, the entire team is notified immediately. Hence, the problem can be easily resolved before someone else uses the corrupt code [32].

By extension, CI helps to reduce tensions and quarrels amongst team members. It also increases developers' levels of confidence of developers to 'break' the code, allowing them to be more productive and enthusiastic [22]. Newcomers will also be able to start the project easier. The quality verification process can isolate, and trace bugs more efficiently owing to the different versions and builds of the code. Finally, CI also makes it possible to deploy projects faster owing to automation [14].

Continuous Delivery follows CI before Continuous Deployment. Continuous Delivery is the logical continuation of CI, where the focus is on ensuring that the code compiles well, and that it is functional, both in terms of production and quality [50], by launching unit tests as regularly as possible. There are also other types of testing, which are as important to ensure code and software quality [54].

However, these tests cannot be conducted without having an environment deployed. In Continuous Delivery the application is built and ready to be released for production at any given time [52]. Conversely, Continuous Deployment is the extension of continuous delivery: deployment is done in an automated way through a pipeline, which determines the steps involved in the build and deployment strategy from start to finish [54]. All compilation steps, unit tests and other automated tests must then be green before engaging in deployment [29].

There are several tools and software dedicated to CI and delivery. In general, organisations base their choice on the cost of the tools, as well as their popularity [29]. The most reputable CI solutions include Jenkins, GitHub CI, Circle CI, Codeship and others. A CI tool serves as a version management and collaboration platform [22].

A version management and collaboration platform facilitates and accelerates the software development process [45]. An open-source code management system is software on demand, namely Software as a Service (SaaS). It allows storing the source code of a project and trails the complete history of all modifications made thus far. The platform provides tools to manage potential conflicts that result from the changes that several developers made [59].

A version management and collaboration platform make it possible to collaborate effectively on the same project. A good version management and collaboration platform should be flexible to accommodate potential changes owing to extensions of functionalities. Also, build and test loads should be distributed on multiple machines [22].

2.4 Software testing

The complexity of software applications is increasing significantly, working frequently in multi-platform and multi-layer environments. Applications are built in conditions that are quick and agile, applying determined requirements [60]. Given this complexity, software testing and SQA have become much more crucial as a way of enhancing the software quality and minimising potential risks [61]. Testing involves running the program with the intention of finding anomalies or defects [62].

The test process is defined to consider the required activities to ensure the following types of tests: planning; analysis; design; implementation; execution; monitoring; reporting on results; and fencing [63]. It is possible to engage in these activities in a technically correct manner without achieving an appreciable result for the company or client. To obtain the correct test results, the developer must know, which stakeholders he/she will interact with each time to be able to have the right input parameters to produce quality outputs [16].

In SQA, testing is a means to verify various attributes of a system and different aspects of its use, whilst ensuring that the product performs as expected, and does not perform any of the functions that it is not supposed to [20]. In other words, testing software means validating its compliance with the requirements in relation to the whole specification, and the design of the software, according to criteria. Some criteria include fields of entry and acceptable error rate, for instance [10].

Hence, software testing is part of quality control and benefits a lot from automation. It includes various techniques necessary to detect software problems. In addition, a further goal of software testing is to ensure that detected bugs are fully fixed without any side effects. Several test strategies can be implemented for a given project. This will depend on the type of system to be tested, and the development process used [62].

It is important to remember that testing is a partial verification method: exhaustive testing of a program by injecting all the possible inputs is generally not possible [18]. There is no proof that a program is correct by testing alone. Testing can only reveal the presence of errors, but never their absence [6]. Testing is primarily utilised to improve the quality of the software, which not only lowers development and maintenance costs, but may also prevent fatal damages [28].

The tests alone do not guarantee quality. Once again, it requires a collaboration between actors to associate the testing process with something that makes sense to the stakeholders. In addition to the test assignments, the test policy should mention how the tests fit into the quality policy of the organisation, if such a policy exists [64]. If no quality policy exists, it is important to avoid any confusion and to explain to the various stakeholders that a good testing policy does not mean that the quality requirements will be met [65].

Testing consists of executing as many software behaviours as possible. The goal is to reduce the number of tests, while maximising their significance by conducting tests, which are able to reveal potential faults [23]. Testing occurs on a dynamic level since the program should be executed as opposed to static methods for analysis, where the correction technique is determined by examining the source [63].

2.4.1 CATEGORIES OF SOFTWARE TESTS

Software testing is a technique used to find faults. Ideally, this search for faults should be conducted at the earliest opportunity and in parallel with development [66]. As Figure 2-5 explains, tests are conducted at several levels, and the practice is known as "test-driven" development. First, it is particularly easy and advisable at the "unit test" level, where the developer writes the tests in parallel with the function that he/she codes, or sometimes before or just after, using the tests to correct bugs encountered until satisfied with the outcome [16].

Integration tests are then used to validate software sub-systems with each other: software-software integration tests or interface between software components; or software-hardware integration tests or interface between software and hardware [61]. These usually happen as soon as a functional sub-system, namely module or object, is fully tested individually via unit tests.

The interface testing usually occurs on a host machine's software or hardware target machine with minimal test bench such as input simulation and output acquisition [36]. The goal here is to ensure that the different entities can work and communicate properly when "integrated".

The test is, therefore, perceived as a means of placing the program in default (test-to-fail) and it naturally leads to obtaining a non-regression test suite [66]. The developer writes the test suites, using the term "white box" test, as the developer not only knows the specifications of the functions, but also of its implementation. The quality of this suite can be gauged by its capacity to capture faults or defects during code evolutions when implementation evolves, though the interfaces and specifications remain the same [16].

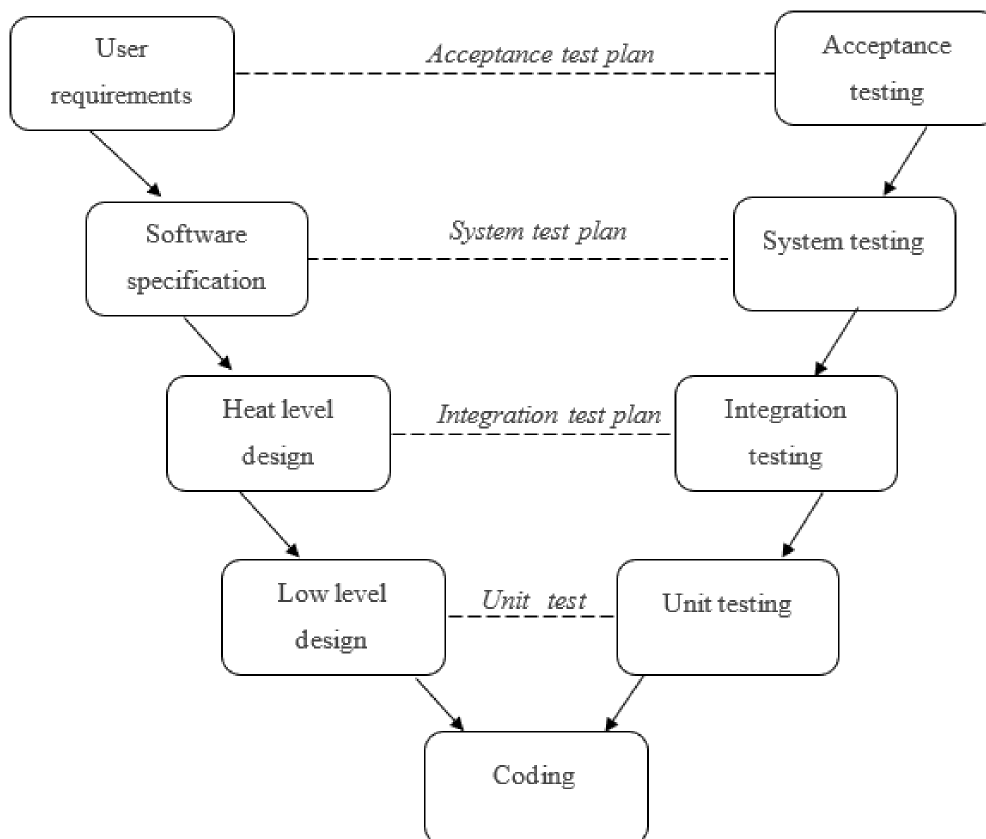


Figure 2-5: Summary of software testing [67]

The test cases can emerge from the specifications alone, namely by the "black box" approach, because the implementation is seen as a black box. In the black box approach, only the inputs and outputs are known, contrary to the "white box" approach, where the test cases are determined by using the source code and knowledge of the internal mechanics of the software.

There are different types of software testing, which can occur at different stages, namely functional, non-functional and structural [36].

A test of the specifications is also called a functional test: the purpose of this test is to explicitly highlight the system's functionality that appears in the specifications. Functional tests reveal the software's conformity mainly in respect of specifications, quality, performance and interfacing [31]. The non-functional aspect, which determines the correctness of the software's usage relates to configuration, compatibility, documentation, and stress testing.

A test from the source code is most often a structural test [62]. The creation of the test case is done based on the execution paths associated with the code, or other structural elements such as seeking to reach an instruction in each memory state. Structural elements relate to the code choice to foresee implementation faults or functions [25].

It is good practice to automate the execution of test suites through evolution of the software development [63]. This is made possible via CI, which is a set of procedures used in the software development process, consisting of verifying each modification of the source code. The goal is to ensure that the result of the modifications does not produce regressions in the developed application [14].

In other words, CI is used to verify that adding or changing lines of code does not introduce errors or problems in the existing code. The main focus of this practice is to deal with integration issues early in development, hence the testing [22].

2.5 PREVIOUS STUDIES

The use of information technology is widespread in all organisations. The software industry tends towards standardisation of the software's lifecycle, and many tools are used to study the software quality. However, many organisations tend to neglect those standards and, as a result, software quality is negatively affected [15]. This is not without consequences as poor code quality is associated with a high number of defects, affecting the quality of the software. In the IT field a single line programming error in the manufacturing process may cause a recall of units, costing the company millions [20].

The increasing need for SQA is significant in the current technological era, as software impacts people's lives daily. The implication of software-based devices is growing in major fields such as medicine, aviation, aeronautics, education, transportation, the military and more.

An example to illustrate the extent of this implication is in medicine, where surgeons practice software-assisted surgeries with the use of robotics [68]. In this case, the life of a patient is at stake and a single mistake or, in this case, defect, could jeopardise it. Therefore, the matter of quality extends beyond bugs and glitches on the interface. Similar studies were performed in the past as means to improve software quality. However, some limitations were observed.

The study conducted by Butt et al [12] focused on software quality standards. It depicts the importance of following software quality standards to guarantee the success of a software project. The risks associated with the lack of quality were not discussed; however, risk management is considered to be part of assuring quality. The standards depend on the type of software, according to the study, as the software properties would determine the required standards and, therefore, generalising standards is not always possible.

However, the authors discussed the set of standards, as defined by the ISO 9001/9000-3, and made use of Capability Maturity Model Integration to determine the basics and baseline in terms of software quality standards. Therefore, in this study SQA and control processes are enforced in a model that depends mainly on the standards of software quality. The study used no tools and techniques, while the steps followed were not outlined in a series of steps to develop a framework.

A study conducted by Owens and Khazanchi [20] reported that less than 30% of IT projects can be considered as being successful. The study identified the causes of software failures, namely improper planning, poor scope time and cost definition, and most importantly, problems regarding quality control. Hence, the study depicted a need for SQA in IT projects and proposed a framework for implementation to control risks associated with poor code quality and to safeguard the software's quality.

Here, the processes involved in SQA are explained in detail and a framework is designed based on the processes. However, the limitations observed in this study include the lack of technological implications, as well as no definition for software quality standards.

A study conducted by Lysne [18] is the closest study that relates to the topic at hand. It provides an overview of SQA practices in the project life cycle. The study detailed the importance of software quality to avoid potential defects and to avoid costs and other problems associated with poor quality software. The focus of this study is to improve the overall quality assurance and software testing processes in the project lifecycle. This includes a detailed background of software tests, automation and other practical tools and techniques that are available through technology.

The result of the above-mentioned study provided a context to improve quality based on SQA principles. However, the shortcomings of the study by Lysne [18] included that the proposed solution is not one that can be adaptable because the definite sequence of steps were not properly defined. The proposed solution is designed for a specific company and the characteristics are adapted to the chosen company. This results in the proposed framework not being generic enough if not at all.

In a study conducted by Salger, Sauer and Engels [17] the focus was on the specification requirements to define the quality gates. The study proposes an SQA framework for computer-based business information systems. The framework in question consider several aspects of quality in terms of standards as well as software testing. However, it is an exceedingly high level, and although the steps involved in the proposed framework are outlined, the processes are relative to the concept of quality gates created and, therefore, cannot be generalised or adapted to other frameworks.

The study by Hossain [15] focused on SQA in terms of automating software testing to ensure quality and defined risks associated with uncertainty on several levels, which need to be avoided. The study provided background relating to risks associated with software defects and other challenges involved in all SQA processes. Therefore, it focuses on the tools and techniques involved in SQA and SQC, namely software testing, as testing is a way of ensuring software quality. The importance of software testing is well defined; however, this study does not provide the steps and a framework for SQA.

The study by Sowunmi et al [69] focused on evaluating SQA, including defining software quality clearly. The study focused on practices and, therefore, outlined some of the processes involved. Although this study does not mention the quality standards and SQC, it does mention the challenges and risks for software quality. However, the study does not talk about mitigating

the risks by implementing software testing, nor does it mention the tools and techniques. Thus, a series of steps is not presented in this study.

The study by Khalane [70] defined software quality, as well as SQA. However, no standards were defined, and it did not mention SQC. Although the study mentioned the shortcomings of software quality in the form of potential risks, it did not mention the processes of SQC. The study discussed s technological implementation in the form of coding and testing techniques but did not provide a series of steps.

Table 2-2 summarises the findings and gaps from literature based on the studies that were found to be closely related to the topic at hand. The table identifies the concepts central to the study along with the main studies to determine the focus of each. The shaded cases indicate that the study covered some aspects of the concept, while the white cases indicate that the concept was not covered by the study. The goal was to identify the gaps found in literature and to derivate the need for the study based on the findings and shortcomings outlined in Table 2-2.

Table 2-2: State of the art matrix indicating gaps in past studies

Source	*Software quality*	*Quality control*	*Quality standards*	*Associated risks*	*SQA processes*	*Software testing*	*Tools and techniques*	*Steps*
[12]								
[20]								
[18]								
[17]								
[15]								
[69]								
[70]								

Previous studies outlined the importance of software quality. In general, these previous studies have properly defined SQA, while depicting its importance in the software development process. The literature further states that SQA goes hand in hand with SQC [18].

However, certain gaps were observed in the literature. None of the studies presented a fully implementable automated and generic SQA framework, which considers all aspects mentioned previously, namely software quality, control, standards, potential risks, SQA processes, tools and techniques and a predefined series of steps to follow. Implementing a proper SQA and SQC process in an organisation without a solid SQA plan or framework, can prove to be detrimental [71].

2.6 SUMMARY

This Chapter provided some background information on SQA and all the concepts central to the study. The importance of SQA and SQC in software development was highlighted. SQA is an important concept in the area of software development. This study aims to propose a framework for SQA.

Providing assurance to the quality of the software is paramount in ensuring that the software product fulfils its purpose. A product which does not satisfy the needs and requirements of its users cannot be considered of quality as quality standards are based on the functional and non-functional requirements. SQA processes need to be present at every stage of the development process in order to ensure the software quality.

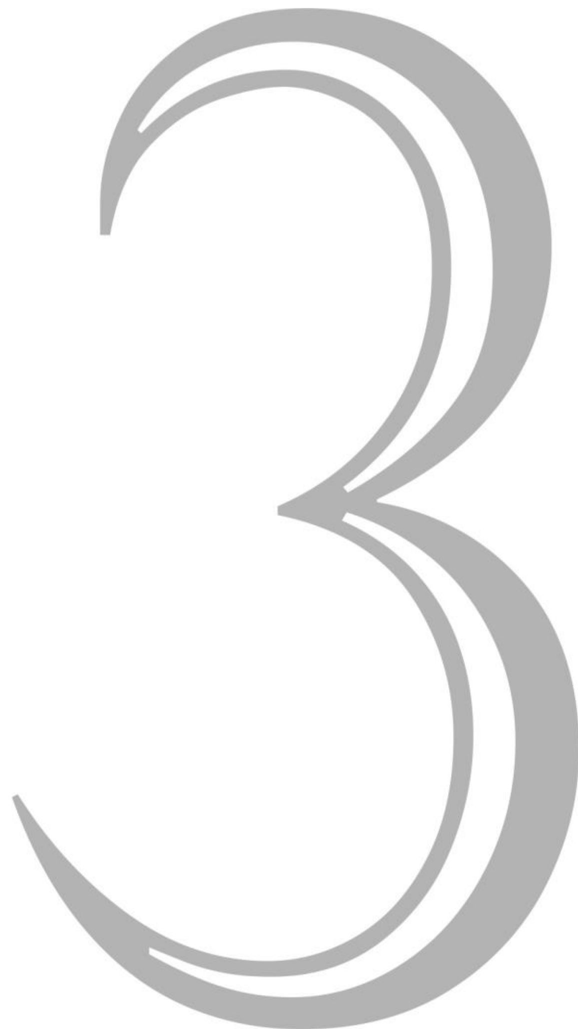
After SQA comes SQC, which must be enforced through the life cycle of the developed product. SQC ensures that the assured quality remains up to standards. This involves a series of iterations and maintenance plans as well as providing support to the users. SQA and SQC both ensure that the risks associated to poor software quality are mitigated and most importantly prevented as SQA is a proactive approach.

In order to be efficient and trustworthy, automation is to be used. Automation includes conducting the development process using CI and enforcing as many tasks as possible to be automated, mostly automated tests. Therefore, the proposed framework is to be generic, automated and enforce the principles of SQA.

The following chapter will discuss the development of the solution. This includes a detailed explanation of the methodology developed and followed in order to develop the proposed SQA framework.

CHAPTER 3

Development of the Solution



3 DEVELOPMENT OF THE SOLUTION

3.1 INTRODUCTION

The previous chapter provided a literature review of SQA and control. A detailed overview was given on software quality, as well as SQA and SQC, and the importance of their implementation in project development. This includes the processes involved, the main concept to implement, as well as the tools and techniques that are available via automation.

Previous studies on the topic were also analysed and limitations were observed, which led to a need for this study. The need for this study is to develop a well-defined automated and generic SQA framework. This chapter discusses the methodology that was developed and used to reach the final solution, which is the proposed framework that addresses the need for an automated generic SQA framework.

The following sections describe each step and the techniques that were implemented to complete each. The methodology was derived from important stages identified in the reviewed literature. The stages involved determining the quality requirements, as well as the steps leading to designing the final framework.

3.2 DEVELOPMENT OF THE FRAMEWORK

The framework consists of two phases: planning and design, as well as implementation. Figure 3-1 illustrates the steps of the methodology that was developed and followed in order to design the proposed framework. An overview of the following steps can be found in from section 3.3 to section 3.7. The steps are as follows:

1. **Understand requirements**: determine what is expected of the outcome or solution.

2. **Investigate the standards**: after identifying the requirements, the next step involves investigating the generic quality standards based on the requirements.

3. **Develop quality checks**: next, the quality standards can be outlined in the form of a checklist that can be referred to when trying to add new code to a repository.

4. **Identify manual processes**: the processes involved in ensuring quality that are typically done manually are identified in this step.

5. **Automate**: because the goal is to create an automated pipeline, this step is crucial and involves automating the manual processes to have the least possible human involvement in the overall activity.

6. **Design workflow plan**: once all the steps above have been individually developed, they can be organised in the form of a workflow plan, which is a series of steps that are presented in an organised manner.

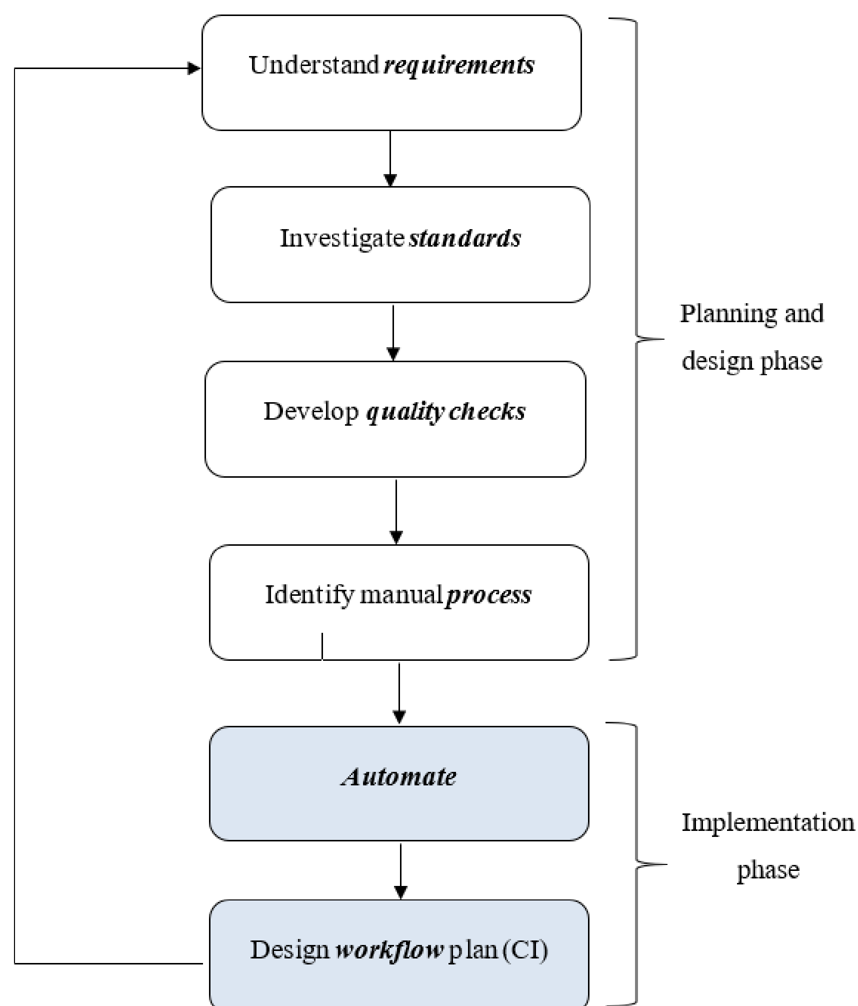


Figure 3-1: Development of the framework

3.3 UNDERSTANDING THE REQUIREMENTS

The first step consists of gathering information about the framework's expectations. Understanding the requirements will ease the process of developing the framework, as well as its implementation. The framework will not fulfil its purpose if it does not meet the requirements.

As explained in Chapter 1, a requirement is a condition that must be met. The requirements determine the factors that the framework should consider in terms of each project. The requirements should also ensure that the framework meets the study's objectives. The goal is to deliver a complete description of the framework, which includes the purpose and core processes that will be focused on, and technical requirements and resources.

It was established in Chapter 1 section 1.3 that the need for the study is to develop a framework for SQA that will respond to the problem statement. The problem identified was the significant risks associated with poor software quality. In order to remedy to this poor quality, the framework should be generic in order to be easily adaptable and automated so as to reduce the amount of potential errors as far as possible.

The proposed framework should also follow an organised set of steps in order to be precise and easily adopted. The framework should account for the agile development environment and be iterative to provide the ability to repeat as needed. As SQA is a reactive process, the framework should be informative for the user in a preventive way.

Therefore, the proposed requirements of the proposed framework are as listed below:

- Generic: the ability to adapt to different projects;
- Automated: making use of automated functionalities to save time while reducing human involvement;
- Preventive: the ability to detect potential errors by implementing automation and performing a series of checks throughout the process; and
- Informative: inform the developer about the process, as well as provide feedback about errors' statuses.

3.5 INVESTIGATING QUALITY STANDARDS

Quality standards, as explained in Chapter 1, ensure that the client's requirements are met on several levels. Therefore, to determine the specific quality standards that need to be met, a thorough investigation should be conducted. This investigation usually involves communication with potential users. The documentation of their needs in terms of the main characteristics of software quality such as usability, adaptability and more should be documented. This happens during the planning and design phase.

However, in the background, other quality standards are also important to consider for software developers. These standards are usually defined within an organisation and can be present in the form of coding standards. Ensuring a good coding practice will have a positive effect on the outcome and will almost always ensure that the visual and usable requirements of the clients will be met. This study focuses on such standards, as the framework aims to account for programming standards and best practice, in general.

Programming standards can be classified into several categories. The coding standards identified in this study are outlined in the Table 3-1. These quality standards are verified during the quality checks.

The coding standards will ensure the quality of the code and should make it easier to avoid problems timeously including bugs, crashes and misfunctioning errors in the system. Readability is also crucial when it comes to code maintenance, as maintainability is a standard of quality. Specific decisions made regarding indents, for instance, do not have major consequences for the solution, but rather for the developers who require consistency in the coding.

The database interaction, as well as well-structured programming will ensure less complexity, which facilitates maintainability once again. It will provide for better delivery and faster execution of tasks on the user's side. With proper error handling, the overall flow of interaction between a user and the application can be improved.

Table 3-1: Coding standards

Programming standard	*Definition*	*Benefit*
Indentation	Special arrangement of the text of a program showing offsets at the margin.	Better reading of the code and better disposition of the lines.
Inline comments	Providing the description of a piece of code, all its public access points, and its dependencies by describing implementation of a class, method, property, or any special variable (and their relationships sometimes), when the need requires.	Serves as a documentation by informing, annotating and explaining the intent of the programmer and provides the necessary information for easy maintenance for the next person to use.
Structured programming	Programming in a structural way in which the code is split into several functions or into sub-tasks or modules, while interacting together.	Separating each task into smaller entities makes it is easy for the programmer to debug and test. Changes can be made easily without modifying the entire solution. It is also easier for code reuse and makes maintainability much easier and efficient.
Naming convention	Refers to the rules and guidelines adopted by members of a software project for writing and formatting code or selecting	Aims to improve readability of the code: must allow the programmer to identify at first glance a maximum of elements in the code to find their way easily, to know where to find things, etc. Once adopted,

	the names of identifiers or elements in the code and documentation.	they greatly facilitate writing, maintenance and help to avoid certain errors.
Errors and exception handling	Refers to the behaviour of the program in the face of an unexpected event, which interrupts the flow of when executing an application.	In the event of an exception occurring, an exception handler is executed, which is a structured way to separate normal logic from that, which handles errors. This increases the readability, maintainability, and robustness of programs.
Database interaction	Defines a mode of communication between the code and the database.	It is advisable to make use of abstraction to separate database queries from the actual code, as it makes it easier to maintain and test for basic operations.

3.6 Development of quality checks

The formulation of requirements and needs includes analysis tasks to determine the needs or conditions that should be met by a new or modified software product. It involves considering the requirements of the different stakeholders of the project, as they may have conflicting needs. The tasks revolve around the analysis, documentation, validation and management of software or system requirements. Needs analysis contributes to the success of a software project, but when done poorly, it can lead to project failure.

Requirements should be documented, feasible, measurable, testable, traceable, and linked to the customer's business needs that have been defined, or to exploit opportunities. It defines a sufficient level of detail for the design of the system. From the requirements, a list of quality checks can be derived. As the programmer completes the piece of code, the checks can be evaluated to ensure that the requirements are addressed.

This usually ensures that the quality standards are met. The checks involve quality verification in a defined order. They can vary from formatting to enforcing an acceptable number of warnings. If a check does not pass, the programmer can then be informed and given an opportunity to effect the necessary changes. This checklist should be generic and checked at each iteration. This should also be the first step every time a developer wishes to incorporate a new piece of code into the master branch, which represents the main source code repository.

3.7 AUTOMATION

The implementation phase of the framework must be automated. When it comes to automation, it is important to determine what and how to automate. As mentioned in Chapter 1, automation has the benefit of reducing the required time for a task to be completed, whilst limiting human error. Therefore, before attempting to automate, it is advisable to first identify the manual steps that may qualify for automation. The following paragraphs discuss some of the potential automation opportunities in software development.

In CI, once the programmer has developed a piece of code, it must be reviewed. This step is typically done manually; however, the automation of some verifications in the code review can be particularly useful and save the reviewing time. A pipeline is a sequence of steps to standardise and facilitate the delivery of software to a given environment.

The pipeline is usually triggered when sending a code to the repository. When enforcing the quality checks in the pipeline, the process will ensure that the quality checks are met before the reviewer can proceed with the actual review. The reviewer can then focus on the actual content and functionality of the code and not waste time on checking for proper indentation, for instance.

Also, when a piece of code is integrated with the rest, it must be thoroughly tested, especially to test the results against the inputs and outputs. According to the section of software testing presented in Chapter 2 section 2.4, in such cases unit tests and integration tests are advisable rather than having a person sit and evaluate each test case for errors. Therefore, implementing a segment to perform automated tests in the pipeline will save time and prevent potential defects in the application later.

When implementing automation, it is necessary to make use of tools and platforms. Tools and platforms support the implementation of DevOps, containerisation and version management and collaboration. The development process and delivery of applications, containerisation and collaborative development are facilitated using the chosen tools and platforms outlined in the following sections.

3.7.1 DevOps platform

Some of the most common DevOps platforms to date include Jenkins, GitLab CI and Azure DevOps [72]. The three platforms have grown very popular in the recent years. However, the chosen DevOps platform for this study is Azure DevOps. The reason being that Azure has shown to provide several advantages that suited the company's needs. Namely, usability, stability, robustness, affordability and simplicity. Also, Azure DevOps provides a variety of tasks and is well maintained and up to date [73].

Azure DevOps platform was launched by Microsoft and offers a suite of services hosted on the cloud. The services include CI and Continuous Delivery, testing and project management boards. Azure DevOps is compatible with all programming languages and platforms, and its services are extensible. The offered services cover the full extent of the development cycle. Each service is open, extensible and works for any type of application, regardless of the infrastructure, platform or chosen cloud [50].

The services provided can be used together for a complete DevOps solution, or with other services. Among them are [47]:

- **Azure Pipelines**: to build, test and deploy continuously on any platform and cloud with any language;
- **Azure Boards**: to plan and track team tasks and then discuss them;
- **Azure Artefacts**: to create, host and share packages;
- **Azure Repos**: to access an unlimited number of cloud-hosted repositories via Git; and

- **Azure Test Plans**: to test and deliver, using a resource kit for manual testing and for exploratory purposes.

When it comes to security, tools like Chef Automate or Azure Policy help to manage infrastructure and provisioned applications to ensure compliance. Combined with Azure Security Centre, for example, these tools limit exposure to threats and help to find and remediate potential vulnerabilities [59].

Azure Pipelines is an Azure DevOps service that is compatible with several programming languages and can interact with GitHub, from which the code can be extracted. Azure Pipelines offers Linux, MacOS, and Windows hosted compilation agents [50].

3.7.2 CONTAINERISATION TECHNOLOGIES

A containerisation technology allows the creation and use of Linux® containers. Some popular containerisation technologies include Rkt, LXC and Docker [74]. The chosen containerisation technology was Docker for its portability, performance, agility, compatibility and most importantly maintainability [73]. The continuous deployment and testing are made easy. Also, the image created in Docker can be used throughout the deployment process, which improves consistency [75].

The Open-Source Docker community is working to improve this technology, which is available free of charge to everyone. Docker Inc. relies on the work of the Docker community, secures its technology, and shares its advances with all users. It then supports improved and secure technologies for its business customers [55].

Using Docker technology, it is possible to treat containers as light and modular virtual machines. In addition, these containers give great flexibility: it is possible to create, test, deploy, copy and move them from one environment to another, allowing for the optimisation of applications for the cloud [59].

Container tools like Docker come with a deployment model that is image-based. This facilitates the sharing of an application between several environments with all its associated dependencies [54]. In addition, Docker automates the deployment of applications within a containerised environment. Built on Linux containers, these tools deliver ease of use and uniqueness,

providing users with unprecedented access to applications, the ability to accelerate deployment, as well as version control and versioning [55].

3.7.3 VERSION MANAGEMENT AND COLLABORATION PLATFORM

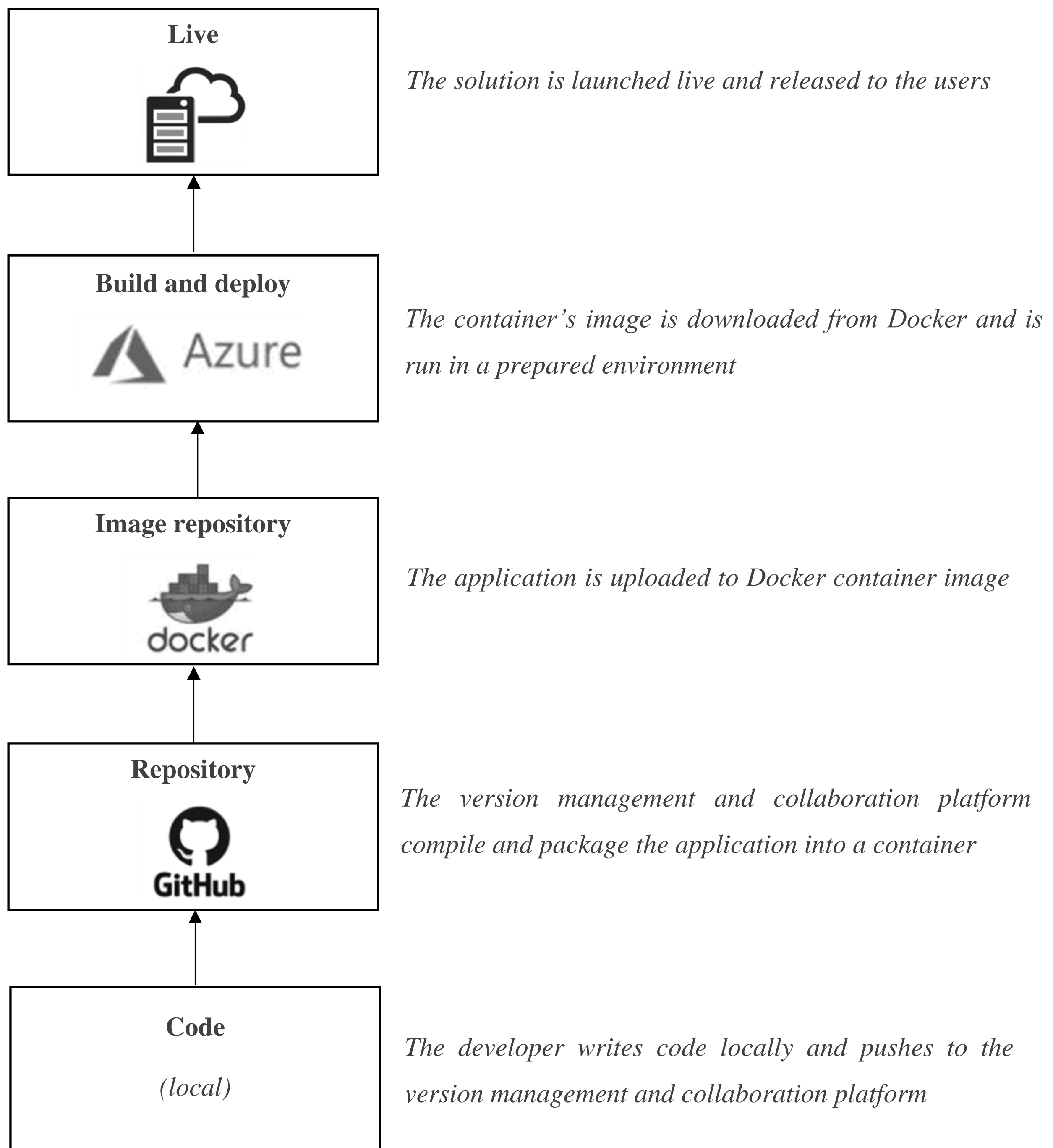
The chosen version management and collaboration platform chosen for this study was GitHub. Some of the most common alternatives are GitLab and BitBucket [76]. However, GitHub has managed to win the heart of a larger number of developers. The reason being that it provides an easier version control, it is free to use, it is scalable and very fast. Also, a very good backup is provided using GitHub. Lastly, GitHub is easily implementable with Azure DevOps [45].

GitHub facilitates collaborative programming by providing an online interface available to monitor the code repository. It also provides administration tools that promote collaboration. GitHub enables its users to modify, adjust, and enhance solutions for public or private use. Each repository comprises of all the project's files, along with information about the history of the revision of each file [77].

In GitHub, developers have three important concepts to remember: fork, which can be defined as a copied repository from different accounts; a pull request, which occurs when a developer wishes to join a new feature to the main repository; and merge, which occurs when the developer's new feature is joined to the central repository [45].

Forks allow a developer to make changes without affecting the original code. If the developer wishes to share some changes, the developer can send a pull request as notification. After reviewing the changes, and if all changes are approved, the original owner can merge the changes into the master repository [22]. GitHub has gained popularity owing to its user-friendliness and the effectiveness of its collaborative version control tools [78].

Figure 3-3 shows a basic summary of the interactions between the developer and operations. Using a GitHub repository and Microsoft Azure to build and deploy, the code is written locally and pushed to the repository. The repository then performs the necessary build operations for the deploy.

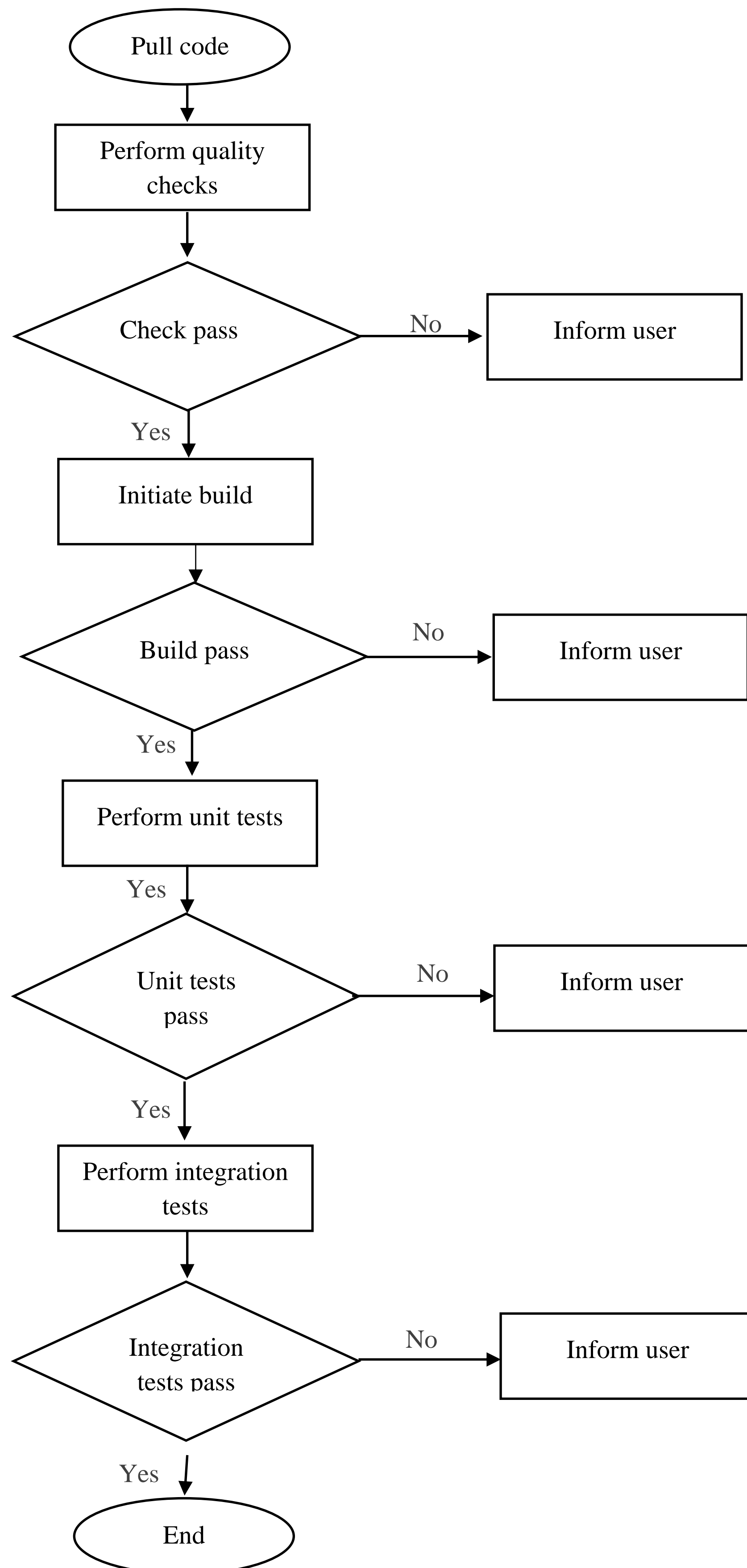

Live
The solution is launched live and released to the users
Build and deploy
Azure
The container's image is downloaded from Docker and is run in a prepared environment
Image repository
docker
The application is uploaded to Docker container image
Repository
GitHub
The version management and collaboration platform compile and package the application into a container
Code
(local)
The developer writes code locally and pushes to the version management and collaboration platform

3.8 Designing the workflow plan

The workflow plan defines all the automated steps involved in the proposed framework in an organised manner, as outlined in Figure 3-3. This workflow plan relies on automation via CI for all the steps performed. All the developer needs to do is push the changes via the integration server and create a pull request. This will then start by pulling the piece of code and perform the automated quality checks.

If the checks are successful it will continue to the next step, which is to initiate the solution's build. Upon successful build, the next step will be to perform the unit tests, followed by the integration tests. Once all the tests pass, the code can be reviewed by the programmer's peers. In the event of any of the steps not passing, it is reported in the form of an error, and the developer can fix the issue and then push again. This happens at each push stage.

The design workflow plan in Figure 3-3 defines the steps that are part of the framework's implementation.


Pull code
Perform quality checks
Check pass
No
Inform user
Yes
Initiate build
Build pass
No
Inform user
Yes
Perform unit tests
Yes
Unit tests pass
No
Inform user
Yes
Perform integration tests
Integration tests pass
No
Inform user
Yes
End

3.9 PIPELINE IMPLEMENTATION

The whole point of building a DevOps pipeline is to be able to work on shorter, repeatable cycles. The objectives are to be able to make changes and modifications quickly, according to feedback from stakeholders and / or users. Any software development process is required to plan, code, compile, test, deliver and deploy the software application. But that is not all, because once in production, the software must be monitored constantly.

The DevOps approach is interesting, as it allows work on shorter cycles with an infinite number of iterations. Hence, the interest is in automating this process and, to do so, it is necessary to choose the right tools. As mentioned in Chapter 2 section 2.3.2, the term pipeline designates a group of software executed in series in a way that the output of one software application serves as an input for the next.

There are different interests in building a pipeline; for instance, chaining a series of software that can write to the standard output and thus avoid creating intermediate files, and adopting a series of tasks that operate one after the other without user intervention.

In its simplest form, the pipeline is merely a chain where each software application participates in the output of the previous one, but there are pipelines where several software depends on the output of another, and vice versa. Some pipelines even contain loops to check if the previous command executed correctly and, if not, restart it instead of launching the next software application. The complexity increases, of course, with the number of stages in the pipeline and the number of connections between them.

The DevOps pipeline is an iterative seven-step process that comprises a cycle from CI to Continuous Deployment. By automating these steps with the appropriate tools, the cycle will be infinitely repeatable [53]:

1. **Plan**: this step allows the team to define functional and operational specifications following customer feedback or simply adding a new feature. The next step is to plan the sprints;

2. **Code**: next, the functionality can be developed using a version manager to facilitate teamwork;

3. **Integrate / Compile**: the code is then compiled to obtain an executable file to deploy in different environments;

4. **Test**: this step is crucial to detect possible bugs or malfunctions, but also to ensure that the application functions correctly in the face of activity peaks or simultaneous connections;

5. **Release**: when all the tests are done and the software is stable, the software can be released;

6. **Deploy**: install the software in the different environments (pre-production, production etc.); and

7. **Operate and Monitor**: finally, this last step ensures that the software and infrastructure are operational, regardless of the environment.

Figure 3-4 shows a sample pipeline implemented, using Azure pipelines integrated with GitHub, where the pull request is created, and from which the source code is retrieved. To make the build execution a little more readable, it is preferable to use features that Azure Pipelines offer, which will allow for the organisation of stages in the pipeline. The stages or jobs comprise several steps, which can be organised, as illustrated below.

For instance, a stage in Figure 3-4 is “Build Quality Check” and the first step at this stage is “Initialize Jobs”. Some steps may be conducted concurrently, while others need to wait for other tasks to be completed before executing. However, each job should be initialised and finalised. Alongside each step is an icon, which indicates success or failure, informing the developer about the status of each task on the job list. The deploy can only occur if all the jobs prior to it are successful.

Jobs

Build, quality checks and uni... 2m 58s
Initialize job 9s
Pre-job: BuildQuality - Build wa... 2s
Pre-job: BuildQuality - Missin... <1s
Checkout Enermanage/ASTIR... 20s
Slack Notification - Build Start <1s
EditorConfig checks (Tabs/Spa... 37s
NuGetToolInstaller <1s
Restore packages 1m 7s
Publish .net core applications ... 19s
Publish .net core applications (... 6s
Publish .net core applications (... 6s
Publish .net core applications (I... 5s
Publish .net core applications ... <1s
Build solution <1s
BuildQuality - Build warnings <1s
UnitTests <1s
BuildQuality - Missing tests <1s
CopyFiles <1s
CopyFiles <1s
Publish Artifact: build <1s
Create build tag (Git) <1s
Cleanup outdated build tags (... <1s
Slack Notification - Build Fail <1s
Post-job: Checkout Enermana... <1s
Finalize Job <1s
Integration Tests 3m 4s
Initialize job 2s
Pre-job: Build quality - Check f... 1s
Checkout Enermanage/ASTIR... 11s
.NET Core 3.1.x 7s
IntegrationTests - ASTIR.BI... 2m 40s
Build quality - Check for missi... <1s
Slack Notification - Build Fail <1s
Post-job: Checkout Enermana... <1s
Finalize Job <1s
Deploy

Figure 3-4: Pipeline implementation

3.9.1 TASK CREATION

Every step in the pipeline is represented by a task which is coded in a .yml file. A .yml file is a YAML format file, which is used as a standard to serialise data. It is used to define workflows in GitHub Actions, and contains build definitions in Azure DevOps. One of the goals of implementing YAML definitions in Azure DevOps is to allow developers to manage their build and to release pipelines as they manage their code with the same tools, following the same habits.

Figure 3-5 shows a sample coded task which represents a step or stage in the pipeline, which is the build quality check for the warnings. This file contains instructions to build, run tests, deploy and so on. Each task has an identifier, a name chosen for display, as well as some input variables. In this case, the identifier is ‘BuildQualityCheck@6’ and the input variable are listed in the ‘input section’. Each task is defined in an organised manner to allow the server to read the file and execute the given task details.

```
# BuildQuality - Build warnings
- task: BuildQualityChecks@6
  displayName: 'BuildQuality - Build warnings'
  inputs:
    runTitle: 'Build warnings'
    checkWarnings: true
    warningFailOption: 'fixed'
    warningThreshold: '5'
    showStatistics: true
    warningFilters: |
      /##\[Warning\](?!.*obsolete.*)/i
    warningTaskFilters: |
      /Build[\s]Solution/i
```

Figure 3-5: Sample task

The tasks are executed at each push. The creation of tasks allows using the CI server to follow an organised set of steps throughout the project, as well as configure the various parameters

concerning its compilation. These sets of parameters are separated into 6 categories, as presented below.

- **General**: the general part allows the project's basic configuration, namely choosing its name and giving it a description. It also allows the user to choose whether to delete the files from previous compilations, enter a GitHub link where the project is located, or deactivate the project temporarily.

- **Source Code Management**: this part allows for selection of the manager versions for the project such as Git or Subversion. This also allows the project's source code to be recovered easily with each modification that is made.

- **Build Triggers:** the build triggers allow the program to trigger the compilation. For example, it is possible to trigger the compilation at fixed times, or at each modification detected on the version that the manager used.

- **Build Environment**: the build environment part allows for configuration of the environment in which compilation is performed. For example, it is possible to delete the files found in the Azure project folder to avoid errors with old files. It is also possible to compile the project in a Docker container.

- **Build**: this is used to define the project compilation rules. For example, it is possible to use an Ant or Gradle script, or simply to use a command lines shell.

- **Post-build Actions**: the post-build actions part allows for configuration of all the operations to be performed once compilation is complete. This is an important part of Azure because it is here that it is possible to generate graphs for the test results and the code coverage, as well as detect errors in the code and use the appropriate plugins.

3.10 VERIFICATION

Verification is required to ensure that the expectations from the pipeline are met. This verification entails that the retrieved information is useful, and that potential problems and errors are detected and reported. Figure 3-6 summarises the tasks execution of a pipeline on a pull request that a developer creates. The information reported here includes time taken for execution, the repository, tests, and coverage, as well as detected errors and potential warnings in relation to the piece of code added.

Figure 3-6: Summary of pull request

The error information is detailed, and a stack trace might be available when clicking on the specific error. This is especially useful for the developer who can see the error's origin, and at

which job or step the error occurred, as well as the file from which it emanated. These errors would not have been identifiable if they were performed manually.

The fact that the build fails when such errors are encountered, prevents the programme from being launched with potential errors. This is an advantage of automating quality checks and tests via CI. If these errors were not detected before the application's launch, some of them could have caused potential threats to the application's well-functioning.

For instance, the errors shown in Figure 3-6 occur owing to failed unit and integration tests. As mentioned in Chapter 2 section 2.4, unit and integration tests ensure that the modules work well as a single unit, and when combined with the correct input and output.

3.11 SUMMARY

This chapter discussed the methodology that was developed to design the solution, which is the framework in the form of the workflow plan suggested. Ensuring quality must be enforced throughout the software development process. Quality standards are first investigated, based on the requirements, followed by a list of quality checks, which is developed from the standards, and typical manual steps are determined. This is done during the planning phase, which is usually followed by implementation.

The steps can then be compiled, and the workflow plan can be designed. In order for the process to be automated, in the background, each task is coded and defined. This is then organised in a file, which will be retrieved and run in an automated mode, using CI. With CI it is possible to observe the progress of projects continuously.

The technologies that were used are GitHub and Azure DevOps owing to their ease of use and benefits with regards to ensuring quality. It is thus possible to detect any problems as quickly as possible and then to remedy them as soon as they appear. Azure's many plugins keep track of the quality of the project because of the easy-to-read reports and generated curves and graphs.

This solution helps to strengthen the quality of these future projects and access to these results by all project members, allowing them to be educated about the quality of their code. In

addition, the use of containers allows the code to be tested in any chosen environment, and isolated from the system.

It then becomes easy to compile the code on different machines. This solution thus facilitates collaboration and the exchange of codes. To continue to guarantee the production of an increasing code quality, the use of other Azure plugins associated with libraries could be considered to detect, for example, compilation errors or syntax specific to each language.

The steps that were considered include the seven stages of a pipeline, and contain checks, builds, tests, and deploys. This is a generic workflow plan that can be used each time a developer wishes to add a new piece of code to the source. The pipeline is specific to each repository. The next chapter discusses the actual implementation, as well as the results that were obtained from implementation of this proposed workflow plan.

CHAPTER 4

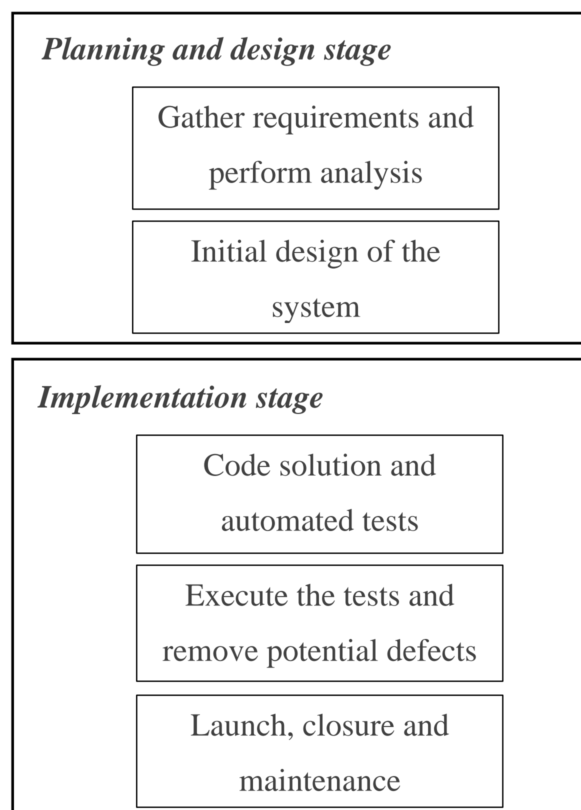
Results and Discussion

4 RESULTS AND DISCUSSION

4.1 INTRODUCTION

A SQA Framework was attained owing to the study's literature review in Chapter 2, as well as development of the solution expounded in Chapter 3. This chapter provides the main outcome of the study, namely the proposed SQA Framework, and serves to highlight the benefits of the developed framework.

Chapter 3 identified the requirements of the framework, which would render it generic and automated. Figure 4-1 illustrates the framework's steps and requirements. This comprises a planning and design stage, as well and an implementation stage. Planning and design involve investigating the potential users' requirements. The implementation phase refers to the coding's technical steps.



4.2 Proposed Framework

The proposed framework combines the design workflow plan suggested in Chapter 3 section 3.7, as well as the requirements gathering. To understand the framework better, Figure 4-2 outlines the legend, which describes the framework's stages and elements. Summarily, the framework comprises actors who perform an action or process, which must be documented and checked. All related actors, actions, decisions, and output documentation are linked via connectors.

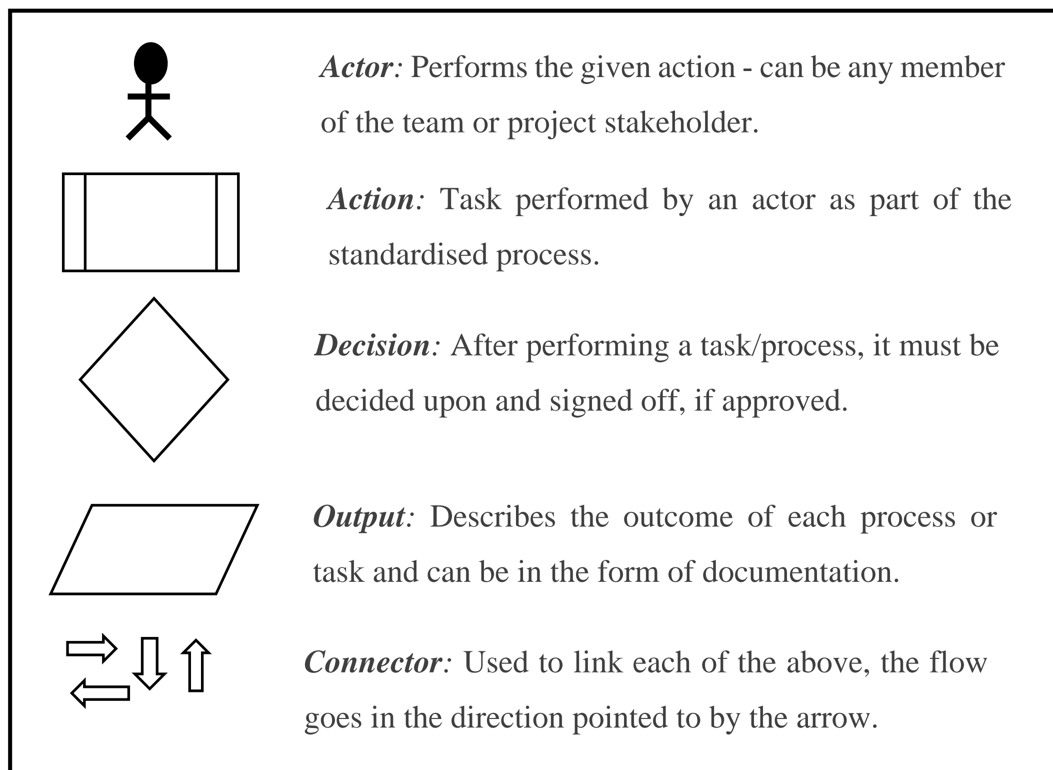
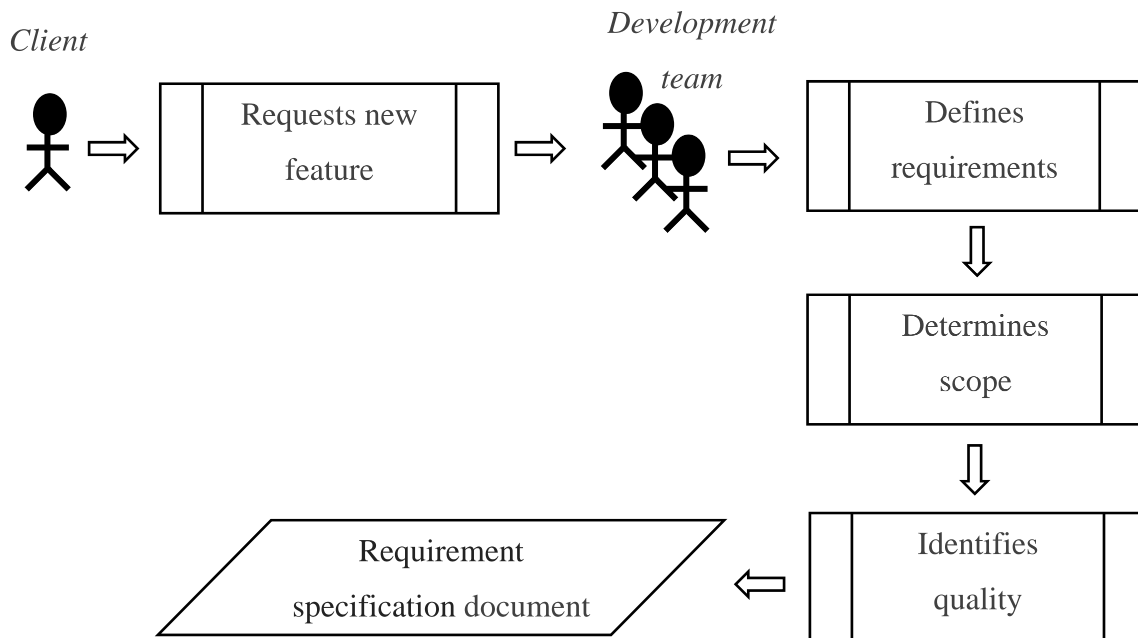


Figure 4-2: Legend of the framework

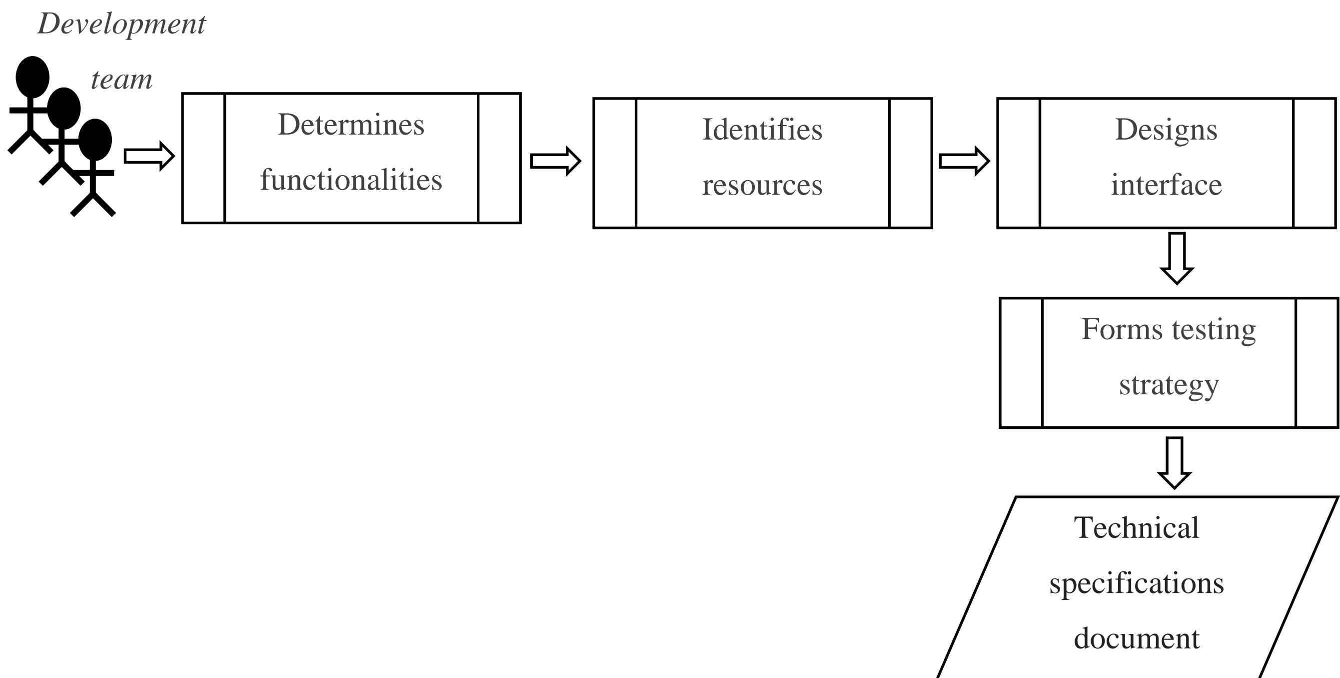
Figure 4-3 outlines the SQA framework, which is the main outcome of the study. The framework follows the legend that was outlined above. Discussion of the framework will follow the figure. As mentioned previously, the framework starts with planning and design, where the actors include all the stakeholders. All the stakeholders determine the requirements, while the objectives are clearly defined and agreed upon at this stage. During implementation, the main actors are the developers. The stakeholders oversee the coding and testing, ensuring that the solution is as defect-free as possible before releasing it to the users.

Planning and design

1. Requirements gathering and analysis



2. Initial design of the system



Implementation

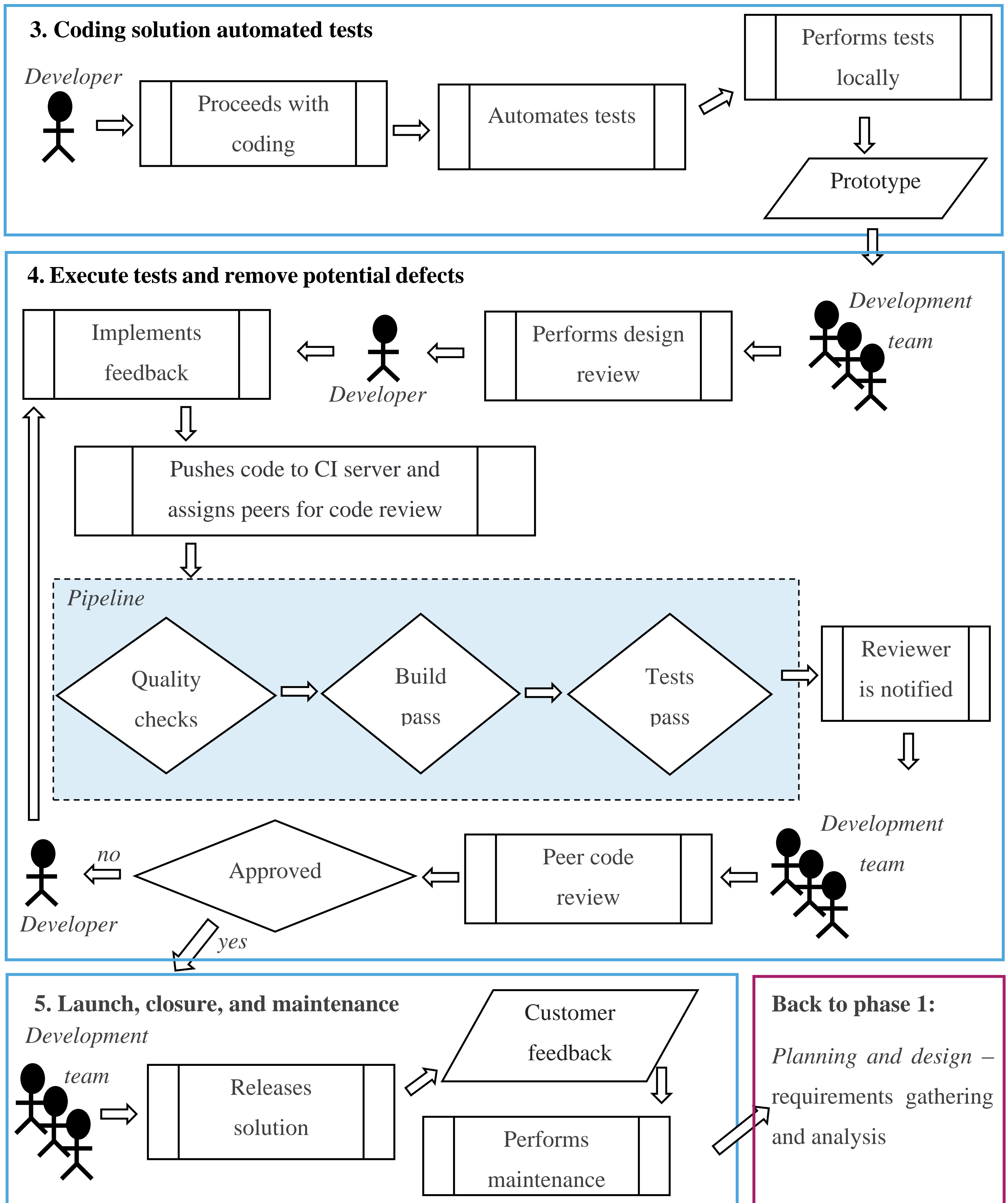


Figure 4-3: Proposed framework

4.2.1 REQUIREMENTS GATHERING AND ANALYSIS

When a client requests a new feature, the development team is mandated to gather information about the project at hand. This information includes all the requirements from a usability and coding perspective. Then the project scope can be defined, and the quality standards compiled. All the gathered requirements should be documented. As the framework is iterative, if requirements are not initially clear, it is possible to repeat this phase as many times as needed.

Clear and well-communicated requirements help development teams to create the right product, which forms the basis for successful product development. A project cannot be of quality if it does not meet the specified requirements. The planning and design stages, therefore, start with identifying requirements.

A requirement contains the most important information about a project. It is a condition or capability that a system must have to meet the terms of a contract, standard or specification, which is formally imposed. The requirements will determine the scope of the project, which determines the objectives. The objectives will, in turn, define the purpose of the project and its inclusion in a global strategy.

The aim is to provide a comprehensive explanation of the software solution that should be developed. The information that is documented includes the purpose, the main business processes that will be supported and the main functionality. As quality was defined as the ability of a system to do what is expected of it, the requirements form the building block of the software quality. The documented requirements can, therefore, serve as a map that provides guidance for the entire development process.

It is always much cheaper to conduct certain modifications in the earlier stage of the development process rather than in the later stages. It would cost more once countless hours and numerous resources and energy have been invested. Having properly defined requirement documents helps to optimise the development process. Task duplication can be avoided, and issues can be structured so that they can be resolved easily.

The main information that should be recorded when identifying the requirements are as follows:

- User needs;
- Dependencies and assumptions;

- System requirements and characteristics;
- Functional requirements;
- External interface requirements;
- System characteristics; and
- Non-functional requirements.

4.2.2 INITIAL DESIGN OF THE SYSTEM

Once the development team has identified all the requirements and defined the project scope and quality objectives, it can start strategizing on the system design. The functionalities of the solution are determined and are based on the identified technical resources. The team may proceed to the design of the interface, while the testing strategy is also defined in this phase.

The output of this stage is the technical specifications, which contain all the information mentioned here. The technical specifications include the programming languages, database requirements and interactions and the choice of platforms and libraries.

4.2.3 CODING THE SOLUTION AND AUTOMATED TESTS

The implementation phase starts with the developer proceeding with the solution's coding. Along with the solution, the developer should compile automated tests, as instructed in the testing strategy. The developer then runs all tests locally to ensure proper functioning of the solution. The output of this stage is the prototype, which is the solution's preliminary version.

4.2.4 EXECUTE TESTS AND REMOVE POTENTIAL DEFECTS

The prototype should be fully tested and reviewed. This occurs when the developer presents the prototype to the other team members to perform a design review. The team members may provide feedback and interact with the prototype to ensure that it operates as it should, and that bugs are found. The developer then proceeds with implementing all the feedback that was received.

Once the prototype is approved, the developer pushes the code to the CI server. The pipeline, as explained in the workflow plan in Chapter 3 section 3.7, is then run. First the quality checks take place, the solution is built, and the tests are executed. If all pass, the code may be reviewed

by the peers and they may suggest how to modify the code to improve the flow and quality. If changes are still requested, the developer must implement them and push to the CI server again. Once the code is approved, it is ready to be released to the users.

4.2.5 LAUNCH, CLOSURE AND MAINTENANCE

Finally, the solution is launched and released to the users. The users are encouraged to provide feedback and to communicate any issue or bug that they may encounter. The feedback, as well as potential updates are implemented during maintenance. The solution must be consistently maintained to ensure that it still serves its purpose and is not outdated.

4.3 IMPLEMENTATION OF PIPELINE

The following sub-sections discuss the results that were obtained following the pipeline's implementation. The results were measured from January prior to the implementation of the pipeline, to October, when the pipeline had been fully implemented for 10 months. The solution was implemented on ASTIR, which is an in-house application system developed in 2014, used for automated translation and information reporting. Active development, optimisation and expansion of features are still ongoing, and the system is in use. ASTIR comprises several in-house applications, which are outlined:

- **Control hub**: ASTIR's headquarter and first application around its creation. It controls all other applications by starting and stopping them when needed. The control hub has about 11956 lines of code;

- **Emailer**: ASTIR's communication application developed in 2015. It is used to send data in email attachments. The data is then placed in the relevant inbox folders. The emailer application has about 2114 lines of code;

- **Converter**: ASTIR's translation application developed along with Emailer in 2015. It translates data from emails, generated by the communicator in the form of attachments in tabular "csv" format, "json" format for logging, and an in-house format called "ist".

It then proceeds to convert the data in the attachments into a more common in-house model. The converter application has about 782 lines of code;

- **Extractor**: ASTIR's data extraction service developed along with Converter and Emailer in 2015. This application retrieves the information from the databases so that it can be imported and exported, as needed, for reporting purposes. The extractor application has about 572 lines of code; and

- **Log maintainer**: Event log service – the most recent application developed in 2019, which is used for events reporting on the system. The log maintainer has about 706 lines of code.

Other ASTIR services are required for importing, exporting, and reporting. Each of the ASTIR application send event logs when performing an action. The types of logs reported are warnings, errors, critical errors, detail, debug, and info. The main ASTIR application logs were analysed to observe evolution of the events on the system over time and reported in the tables below. The months reported on range from January to October 2020. The specific events that were reported include errors, critical errors, and warnings on the system over time. Appendix B provides a detailed version of event logs that were captured in the form of screenshots.

To analyse the verification time on the developers' side, the work logs of five developers were also analysed anonymously. The following sections provide the results that were obtained. The results were used to determine the efficiency of the framework, as well as whether the overall quality of the software improved as a result.

4.3.1 DEFECTS REDUCTION

The first and main attribute that was observed was the overall defect reduction. As explained in Chapter 2 section 2.2.3, a defect in the form of bugs errors, or crashes may present a potential risk to the application as they may prevent the system from functioning as it should. Therefore, defect reduction should be a point of focus of proposed framework. Defect reduction refers to the reduction of errors and critical errors in the system over time. On ASTIR, defects are reported in the form of errors and critical errors. Table 4-1 provides an overview of the evolution of errors that was reported on the system.

Table 4-1: Error reduction over time

Application Name	*Jan*	*Feb*	*Mar*	*Apr*	*May*	*Jun*	*Jul*	*Aug*	*Sep*	*Oct*	*%*
Control hub	236	223	222	214	209	194	186	171	152	143	**39%**
Convertor	132	127	118	113	109	101	92	92	89	82	**38%**
Log maintainer	108	97	98	85	81	78	76	69	70	65	**40%**
Extractor	213	185	179	171	165	164	164	158	141	126	**41%**
Emailer	785	764	648	612	589	584	502	489	481	469	**40%**
TOTAL	**1474**	**1396**	**1265**	**1195**	**1153**	**1121**	**1020**	**979**	**933**	**885**	**40%**

Table 4-1 indicates that the number of errors decreased for each application from January to October. The percentage (%) column shows the overall percentage of reduction from January to October and was calculated by deducting the number of errors obtained in October from the number of errors in January. The column shows the overall number of errors has decreased by 40%. Therefore, there were 40% less errors that occurred on the system from January to October. Figure 4-4 illustrates the results obtained above.

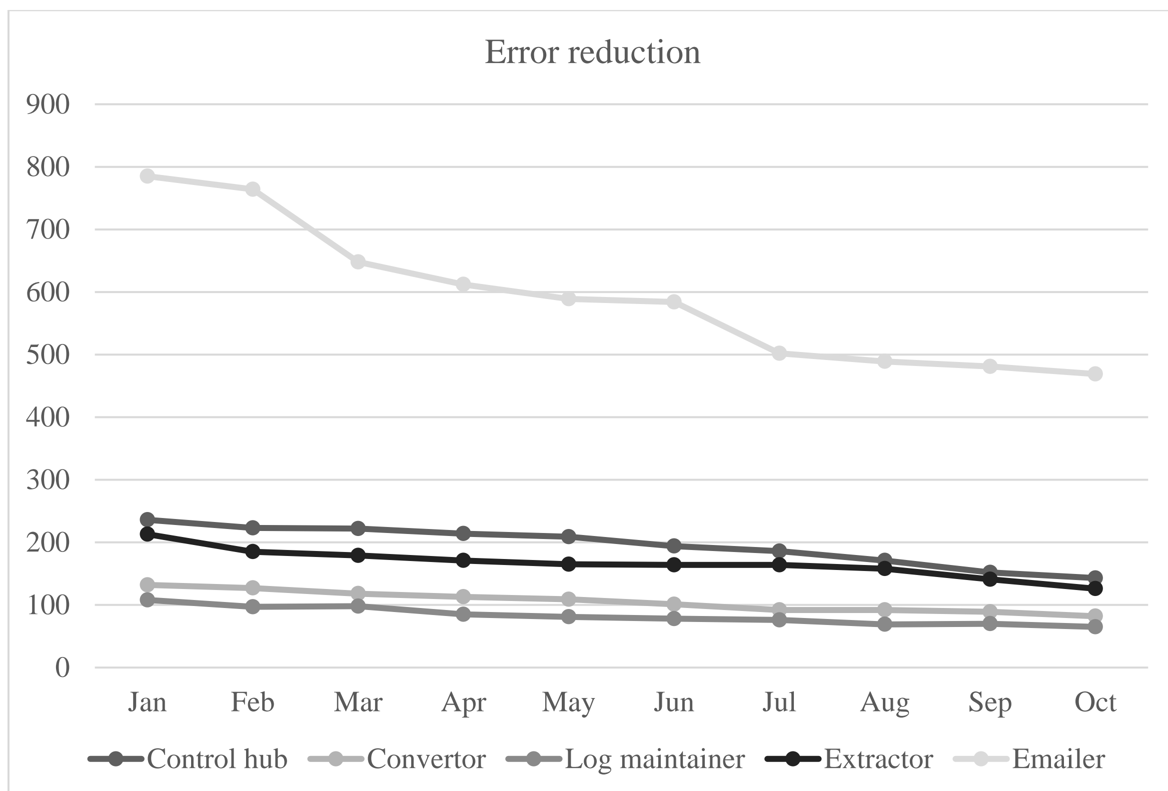


Figure 4-4: Error reduction over time

Table 4-2 provides an overview of the evolution of critical errors reported on the system.

Table 4-2: Critical error reduction over time

Application Name	*Jan*	*Feb*	*Mar*	*Apr*	*May*	*Jun*	*Jul*	*Aug*	*Sep*	*Oct*	*%*
Control hub	157	147	131	125	121	119	112	105	101	98	**38%**
Convertor	78	71	71	69	67	61	59	54	49	45	**42%**
Log maintainer	69	64	61	58	47	49	42	42	40	39	**43%**
Extractor	95	96	91	89	86	82	71	69	63	57	**40%**
Emailer	409	369	301	289	281	273	271	269	264	256	**37%**
TOTAL	**808**	**747**	**655**	**630**	**602**	**584**	**555**	**539**	**517**	**495**	**40%**

It can be noticed in Table 4-2 that the number of critical errors decreased for each application. The percentage (%) column shows the overall percentage of critical error reduction from January to October and was calculated by deducting the number of critical errors obtained in October from the number of errors in January. The column shows the overall number of critical errors has decreased by 40%. Therefore, there were 40% less critical errors that occurred on the system from January to October. Figure 4-5 illustrates the results obtained above.

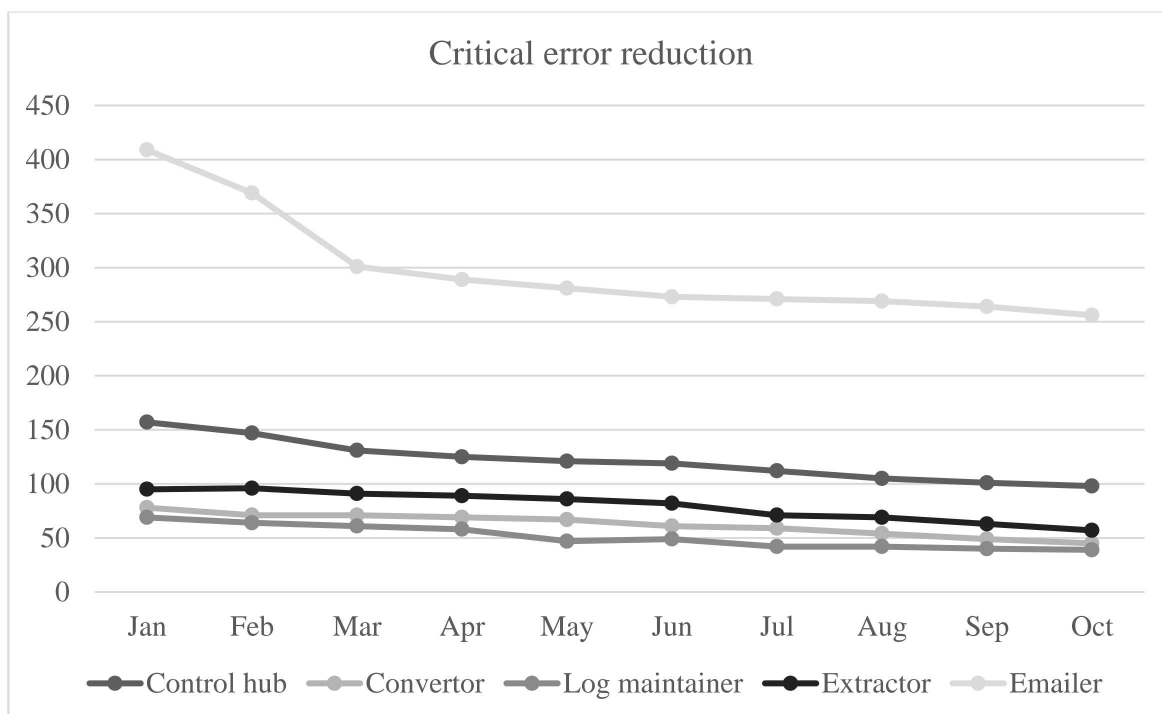


Figure 4-5: Critical error reduction over time

A steep decline was noticeable from January to March in respect of both the critical error and error reductions. Between January and March, major emphasis was placed on reducing errors on the system by enforcing the implementation of automated tests. It became a requirement that each developer should add the required test to each piece of code that was added.

In addition, the steep decline was noticed on the Emailer application. It is the second largest application and most of the errors occurred on this application. Thus, more emphasis was placed on error reduction for this application.

However, it was also observed that after the initial steep decline, the number of errors decreased over time. This can be attributed to the pipeline's implementation. Provided that all the other applications' number of errors decreased over time, it is safe to assume that the pipeline assisted with error reduction over the 10-month period when the study took place.

4.3.2 INCREASED DEFECT DETECTION

Defect detection is another benefit of the proposed SQA framework. Defect detection involves detecting a potential defect before it occurred. This assisted developers to take preventive measures to prevent the event from happening, or at least limit the consequences in case it happens. If more defects are detected beforehand, less defects will occur on the system.

In terms of the analysed system, the detections of potential problems are reported in the form of warnings, which are messages sent by the system, to inform the developers that something is potentially wrong malfunctioning or risky. The warnings occur when an unusual situation happens, or a potential threat is detected. Table 4-3 provides an overview of the evolution of the warnings reported in the system.

Table 4-3: Defect detection over time

Application Name	*Jan*	*Feb*	*Mar*	*Apr*	*May*	*Jun*	*Jul*	*Aug*	*Sep*	*Oct*	*%*
Control hub	10	8	9	10	9	10	10	11	13	14	**40%**
Convertor	19	20	18	18	19	21	21	22	22	26	**37%**
Log maintainer	25	25	24	26	27	28	28	32	33	39	**56%**
Extractor	101	102	114	118	120	125	128	130	135	146	**45%**
Emailer	203	198	201	215	219	217	221	231	243	251	**24%**
TOTAL	**358**	**353**	**366**	**387**	**394**	**401**	**408**	**426**	**446**	**476**	**40%**

Table 4-3 shows that the number of warnings increased for each application from January to October. The percentage (%) column shows the overall percentage of defect detection from January to October and was calculated by deducting the number of critical errors obtained in January from the number of defects detected in October. The column shows the overall number defects detected has increased by 40%. Therefore, there were 40% increase in defect detection on the system from January to October. Figure 4-6 illustrates the results obtained above.

It can be noticed that the total percentage varied significantly from applications and Log maintainer was the highest with 56% whereas Emailer was only 24%. This is due to the fact that Log maintainer is ran along with each other applications whereas Emailer runs twice a day. This makes for more reporting on the Log maintainer side.

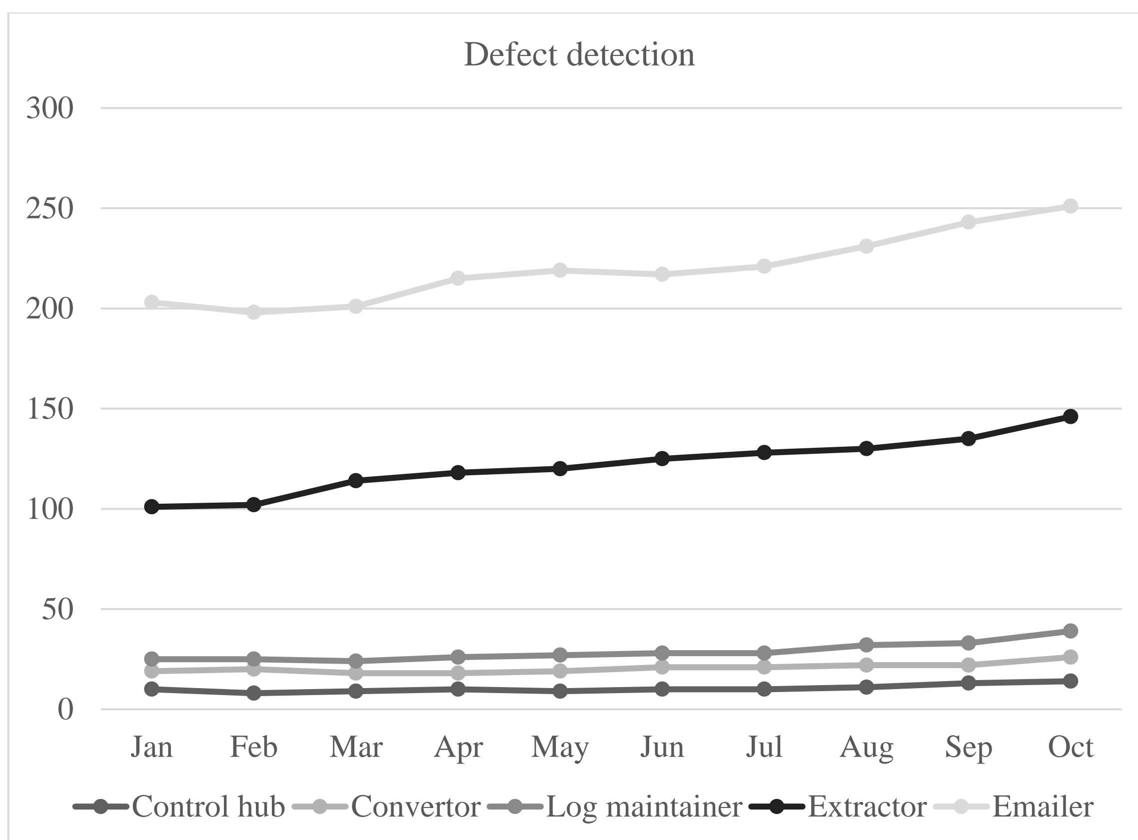


Figure 4-6: Defect detection over time

4.3.3 REDUCED VERIFICATION TIME

One of the main benefits of automation is the decreased time spent on tasks compared to those done manually. One such task is the verification time, which is the time that is required to

perform verification. With the automation of the verification of some coding standards, the developers may spend less time on verification, leading to a reduced overall verification time.

The worklogs are used to keep track of each employee's work tasks, as well as time spent to do these. It is a general requirement for each employee to complete worklogs each day. The worklogs of the developers were obtained and analysed to identify a trend in the verification time. Table 4-4 provides an overview of the evolution of the verification time for five developers, based on their worklogs.

Table 4-4: Verification time

Application Name	***Jan***	***Feb***	***Mar***	***Apr***	***May***	***Jun***	***Jul***	***Aug***	***Sep***	***%***
Developer 1	162.5	154.5	148	139.5	134.5	131	129.5	125	121.5	**25%**
Developer 2	35	32.5	31	31	30.2	31	29	28.5	27	**23%**
Developer 3	27	26	24.5	23.5	23	23	22	21.5	20.5	**24%**
Developer 4	97.5	95	92.5	89	86.5	82	79	75	73	**25%**
Developer 5	147.5	142.5	140	136.5	132.5	128	123	114.5	110	**25%**
TOTAL	**469.5**	**450.5**	**436**	**419.5**	**406.7**	**395**	**382.5**	**364.5**	**352**	**25%**

Table 4-4 indicates that the time spent on verifying errors decreased for each developer from January to September. The percentage (%) column shows the overall percentage of verification time from January to October and was calculated by deducting the number of hours obtained in October from the number of hours in January. The column shows the overall verification time has decreased by 25%. Therefore, 25% less time was spent on verification from January to September.

The time saved indicates that more resources can hence be used better. Figure 4-7 illustrates the results obtained above. The large difference in the time verification between developers is due to the fact that the chosen developers are from different teams, where the volumes of code differ by more or less, depending on the changes requested and the size of the application worked on. However, this does not impact the research or results as the results are converted in percentages.

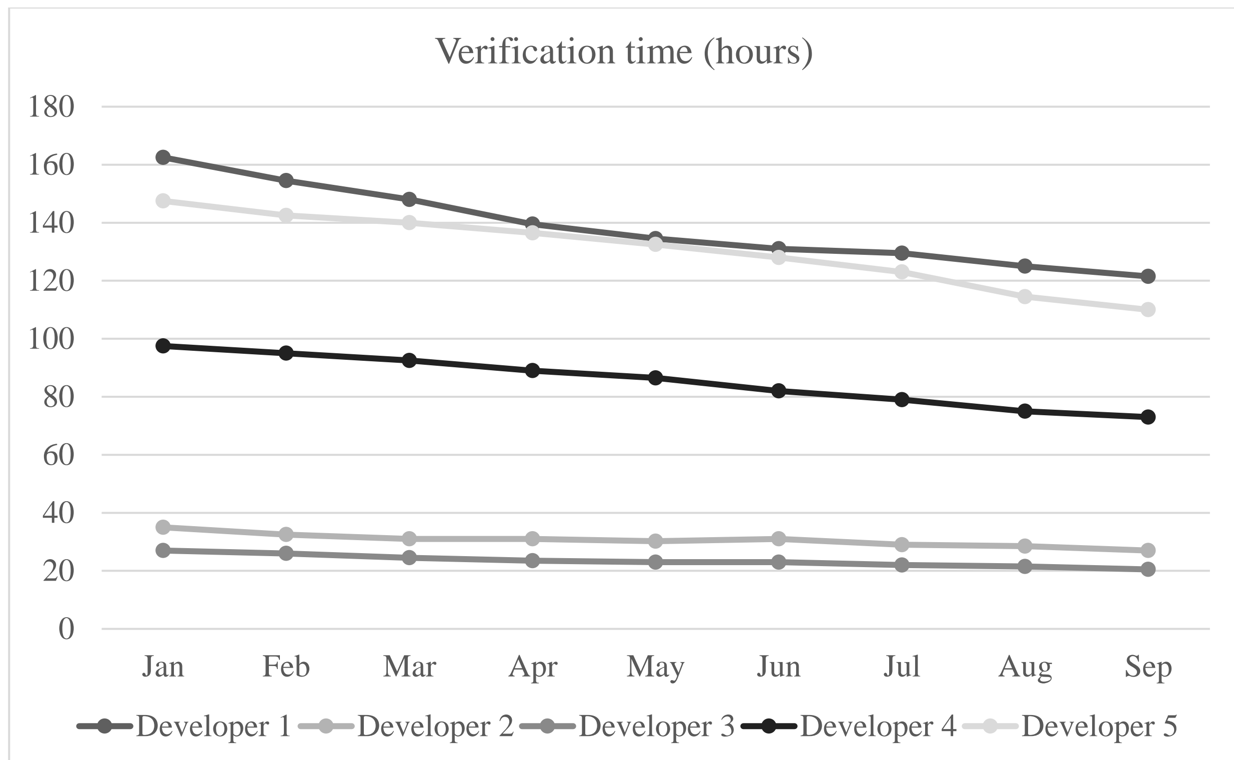


Figure 4-7: Verification time

4.4 ADDITIONAL BENEFIT

Another additional benefit was the increase in test coverage, which is the portion of code covered by automated tests. Implementation of the framework has enforced software automated software tests, in addition to the initial emphasis on the automated unit and integration tests mentioned before.

Code coverage is associated with code testing and helps to ensure that tests run through the entire program that is tested. The tests make it possible to verify that the results that were obtained are indeed what were expected; in other words, ensuring that a piece of code functions as it should. Code coverage is associated with code testing and helps to ensure that tests run through the entire program that is tested.

The tests make it possible to verify that the obtained results are those, which were expected. Code coverage allows for measurement of the parts of the code that were tested. Indeed, conducting tests, even if they are completely correct, is of little use if they only apply to a tiny

part of the project code. Thus, the combination of quality tests, coupled with the measurement of their coverage, ensures that the entire program has indeed been tested.

Figure 4-8 shows a sample test coverage summary on the system in January before implementation using Visual Studio. It can be noticed in the Covered (% Blocks) column that only less than 1% of all files were covered by automated. This means that automated tests were not being implemented at this point in time.

Code Coverage Results

saboozoo_SABOOZOO_2020-01-10.09_09_52

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
saboozoo_SABOOZOO_2020-01-10.09_09_52.coverage	55205	99.91%	49	0.09%
devlib.datastore.dll	19774	100.00%	0	0.00%
devlib.datastore.tests.dll	22441	100.00%	1	0.00%
devlib.logger.dll	489	100.00%	0	0.00%
devlib.testing.dll	758	94.04%	48	5.96%

Figure 4-8: January test coverage summary

Figure 4-9 shows the test coverage following data collection in November. These test coverage summaries show the number of packages and files that were covered and uncovered by the tests. It can be seen in the Covered (% Blocks) column that more than half of the total number of files are now covered by automated tests. This shows that the implementation of automated tests has been enforced and is growing significantly. Appendix B provides the details of the code coverage across all the application by enumerating the number of code blocks covered by automated tests in each of the files.

Code Coverage Results

saboozoo_SABOOZOO_2020-11-10.14_58_21

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
saboozoo_SABOOZOO_2020-11-10.14_58_21.coverage	21094	41.78%	29390	58.22%
devlib.datastore.dll	7083	40.35%	10469	59.65%
devlib.datastore.tests.dll	1368	6.93%	18375	93.07%
devlib.logger.dll	488	100.00%	0	0.00%
devlib.testing.dll	299	37.10%	507	62.90%

Figure 4-9: November test coverage summary

Figure 3-8 and Figure 3-9 indicate a significant increase in the overall coverage. In January, the total percentage of files covered by tests amounted to 0.09%, whereas the uncovered files totalled 99.91%. In November, the total percentage of files covered by tests amounted to

58.22%, while the uncovered files comprised 41.78%. This meant a 58.13% overall increase in test coverage.

Combined with unit testing, code coverage is essential to ensure code quality, thus ensuring the software quality. The obtained results can make it possible to detect a change in habits in the code sooner rather than when it may be too late. Some of the factors that cause such change in habits include:

- The quality of unit tests decreases as recent tests do not properly scan the added code;
- Failure to write unit tests; and
- The arrival of new employees with different habits who have not written unit tests.

It is, therefore, easier to quickly detect and resolve bad habits to maintain a quality code throughout the project.

4.5 Outcome

4.5.1 Validation of the framework

The proposed framework covers all the SQA processes outlined in Chapter 2 (Figure 2-1). The framework comprises a research and analysis phase, where the requirements are identified, and where analysis is performed in respect of the objectives. The implementation phase, where the initial design is created, allows for a design review to be performed based on it. This prototype can then be tested for proper functioning and to remove as many defects as possible.

The prototype is hence refined until the final code is ready for review. During review the team member can suggest relevant inputs about how to improve the coding to avoid potential errors. Thereafter, the developer fixes the code accordingly until the solution is ready for production. As the solution is released to the market, the user can interact with it and constant improvement can be made in the form of support.

The study proposes a generic and automated SQA framework. The goal is to respond to the problem statement in terms of how to minimise the number of defects, as well as to reduce the time consumed on a manual verification process. Based on the results obtained in this chapter, it can be seen that the number of errors was reduced throughout implementation. The number of defects that were detected also increased.

The verification time also decreased, which means that the automated process is more time efficient. In addition, all ASTIR applications were equally analysed, making it possible to implement the solution on several different applications. Therefore, the solution is adaptable to different scenarios, rendering it generic. Consequently, theoretically, it can be validated that the proposed framework responds to the study’s need.

4.5.2 CODING ACTIVITY

A significant amount of coding occurred between the dates that were observed. Table 4-5 provides details of the number of additions and deletions from January to October 2020. These results were obtained from the version management and collaboration platform, GitHub, which shows the coding activity. The coding activity was reported in the form of the number of commits that were made.

Several contributions were made from January to October. It is also worth noticing that the number of additions decreases over time. This can be attributed to the following factors. First, a large portion of old legacy code was removed, reducing the maintenance on that area. Second, with the number of errors decreasing less bug fixes were required. The two factors resulted in a reduction of additions over time. Appendix C illustrates more insight into the contributions provided by the version management and collaboration platform.

Table 4-5: Coding activity from January to October 2020

Month	*Additions*	*Deletions*
January	3596	3956
February	19804	19864
March	2606	2042
April	1814	1273
May	1966	795
June	2735	1270
July	1361	280
August	880	1663
September	518	764
October	682	832

4.6 Summary

This chapter provided the results and outcomes of the study. The purpose of this study was to develop a framework for SQA. Therefore, the main outcome of this study is the proposed framework for SQA. The framework was developed following the methodology that was developed and outlined in Chapter 3. The framework was then presented and implemented in this chapter, reporting results obtained from its implementation.

The framework differs from traditional software development methods, which for the most part contain more manual activities. An automated process is put in place to reduce as much human errors as possible and cutting down on development time. Also, the proposed framework differs from the solutions found in literature outlined in Chapter 2 section 2.5 as it is generic, easily implementable and more importantly, comprises of all aspects of SQA and SQC.

The proposed framework was developed with the aim of assuring quality at every phase of the development process. This differs from traditional methods where the emphasis is to deliver big features as quickly as possible. This framework encourages the delivery of smaller features continuously.

With the proposed framework, the features can be broken down into several iterations owing to the agility of the development process. Releasing a smaller feature at the time reduces the potential risks of defects proactively and facilitates traceability and maintenance. All in all, the aim of the framework is to facilitate the development of software features that provide quality and satisfies the specified requirements.

The framework comprises several steps, which can be divided into two categories, namely planning and design, and implementation. During the planning and design phase, the users' requirements were investigated, and a checklist developed based on the standards. This checklist contains all the quality standards that need to be met by the final product, as well as the piece of code.

The implementation stage consists of coding, testing, and integrating the piece of code together with the rest. In an agile environment this can be done in iterations as smaller pieces of code are merged each time. Therefore, it is important to ensure that the piece of code that is merged will work with the overall solution, which means that the quality will be maintained, while limiting the number of errors.

The proposed framework provides structured results in defects and verification time reduction as well as an increase in defects detection. In other words, as anticipated, the number of errors, as well as critical errors, decreased on the system by 40%. Also, the number of defects detected in the form of warnings increased by 40%, making it possible for the developers to take preventive measures in terms of the occurrence of defects. The verification time also decreased by 25% as the automation became more dominant on the system, allowing more resources to be allocated on more important activities. This shows an improvement of the software quality, making the system more valuable and reliable as a company asset.

CHAPTER 5

Conclusion and Recommendations

5 CONCLUSION AND RECOMMENDATIONS

5.1 CONCLUSION

Given the current technological race, an important amount of code is generated daily. When developing software products, more emphasis is often placed on innovation and new functionalities, while the code quality is often neglected. Hence, defects may result, jeopardising the overall quality of the software.

To mitigate the risks associated with poor software quality, this study's objective was to develop a SQA framework. The framework needed to follow the principles of SQA and be automated and generic. Therefore, a framework for SQA was proposed. It is a framework that would fit with the project's life cycle in a generic manner, whilst incorporating as much automation as possible.

To develop the framework, the following steps were followed:

- Understanding the framework's requirements;
- Investigating the quality standards;
- Developing quality checks for the projects;
- Identifying typical manual processes;
- Automating identified processes; and
- Designing the workflow plan of the automated steps with CI.

The framework comprises two phases, namely the systems analysis and implementation phases. During the analysis phase, the functional, non-functional, and technical requirements of the project were defined, and the initial design was drafted. Resources were identified, and a testing strategy was developed.

The implementation phase involved coding the solution and automated tests, whilst pushing the code to the CI server to enable the pipeline to run. All the quality checks' build conditions and tests were triggered, and the developer was informed of the process. If anything failed, the developer was informed of the problem/s in detail and could fix them before pushing the changes again. Warnings were also provided as feedback.

The code was then submitted for peer a review. The goal was for peers to identify potential problems with the code and to suggest better alternatives. Once all reviewers approved the code, the solution was launched for users. Feedback was provided and the development team took charge of maintenance of the solution on a regular basis to continue to ensure quality.

Using CI, it is possible to observe the evolution of projects continuously. It is thus possible to detect any potential problems and to remedy them as soon as they arise using a DevOps approach. The chosen technologies to implement the pipeline were Azure DevOps and GitHub. These chosen technologies' many plugins monitored the quality of the project owing to the easy-to-read reports, and the curves and graphs that were generated.

In addition, the use of DevOps allowed the code to be tested in any chosen environment and even when isolated from the system. The image system compiled the code on different machines without having to worry about different installations of libraries, which were necessary for its operation. This solution thus facilitated collaboration and exchange codes.

When implementing the proposed framework, the metrics that were observed were the number of defects in the form of bugs, errors and critical errors and the number of warnings on the system. The work logs of five developers were also analysed to determine improvements in the verification time.

The results showed that the number of errors and critical errors decreased by 40%, which meant that the number of defects had been reduced overall. The number of warnings increased by 40%, which meant that more defects could be detected beforehand. This detection allowed the developers to take preventive measures. Also, the time spent on code verification, based on the worklogs, had decreased by 25%, allowing more resources to be freed up for better use.

Therefore, the framework fills the gap identified in the previous studies. The proposed SQA framework presents a fully implementable automated and generic SQA framework, which considers all aspects mentioned previously, namely software quality, control, standards, potential risks, SQA processes, tools and techniques and a predefined series of steps to follow.

However, similarly to the steps identified where present in previous studies, the steps and phases presented in the proposed framework remain on a higher level. The reason for keeping a minimal level of detail is to ensure the framework remains easily implementable. The steps and phases presented in this study form the general steps that should be present in a SQA

framework. Details vary depending on the organisation, hence the general steps can be adapted and enlarged as needed.

In respect of the study's results, it can be concluded that the proposed SQA would help the development team in its efforts to strengthen the quality of future projects. Hence, the need for the study was, therefore, fulfilled.

5.2 LIMITATIONS OF THE STUDY

A major limitation encountered in this study was time, as it would have been ideal to test the implementation for a few more months. Extending the timeframe would have confirmed that the results that were obtained emanated from implementation of the framework.

Another major limitation of the study was the lack of quality metrics to confirm that the software's quality had improved overall. Also, it was unfortunately not possible to measure the rate at which developers were solving errors prior to the implementation of the framework to provide a more accurate representation of the results. The information was no longer available.

The recorded defects were accrued over numerous months of the scope of the work and were only precise at the recording time. Potential glitches could have occurred after, or potentially compromised functionalities may not have been used until months later. Also, the case that a piece of code could have been rewritten multiple times was not considered. Several residual errors and defects were fixed during the first months, which can potentially add to the reduction of defects.

The detection of defects through peers or design reviews was not recorded, although these reviews are also part of the framework. The focus of the framework was only on automation; however, defects detected manually could have also contributed to an overall reduction of defects on the system.

Another important limiting factor was the difference in each of the teammates' levels of experience and professional background. Some team members may have performed better and contributed more than others. The number of defects detected was attributed to the individual contribution of each developer, tester, and reviewer. In addition, the number of file changes on the system in terms of additions and deletions was not the same throughout the months.

5.3 RECOMMENDATIONS FOR FUTURE WORK

The first recommendation for future work related to this study is to analyse the impact of implementing a framework that focuses on the quality of the code metrics. A study to improve the code quality could eventually benefit the software quality overall.

The second recommendation is to enlarge the time frame of the implementation. The early months cannot provide completely accurate results, while developers were still in the process of adopting the implementation. In addition, to get a better estimate of the impact of the framework in the future, it is recommended to measure the rate at which developers solve problems before the implementation of the framework. This would make it easier to compare the final results more accurately.

The third recommendation is to perform a comparative study of the proposed framework versus other existing frameworks such as URDAD and the contracts framework for mocking & testing. The results obtained would help determining the efficiency of the proposed framework. Also obtaining input from the users' perspectives to determine if their expectations were met using the different frameworks. User experience and user computer interaction also contribute to software quality.

The fourth recommendation is an important suggestion for future work, which is to implement a software test framework to conduct a predetermined test plan. This test plan will assist each developer with new code. It will determine, which test should be written and, which is suitable for the given scenarios.

It is also advisable to implement the framework at another company. Different coding languages and technologies at different organisations would assist to assess to what extent the solution is generic. It is not expected that different results will be obtained. The adoption of the framework in different industries will validate and verify that it has indeed met all expectations.

Lastly, it would be beneficial to develop a small system using the framework and one without the framework in order to provide a comparison between the two. The two systems should be developed by two different development teams with team members who have the same skills level. The results obtained in the final product will compared in order to evaluate the number of defects in the two applications.

REFERENCES

[1] P. Bowen, K. Cattel, K. Hall, P. Edwards, and R. Pearl, "Perceptions of Time, Cost and Quality Management on Building Projects," *Aust. J. Constr. Econ. Build.*, vol. 2, no. 2, pp. 48–56, 2012, doi: 10.5130/ajceb.v2i2.2900.

[2] A. Javed, "How To Improve Software Quality Assurance In Developing Countries," *Adv. Comput. An Int. J.*, vol. 3, no. 2, pp. 17–28, 2012, doi: 10.5121/acij.2012.3203.

[3] H. Zhang and S. Kim, "Monitoring software quality evolution for defects," *IEEE Comput. Soc.*, vol. 27, no. 4, pp. 58–64, 2010, doi: 10.1109/MS.2010.66.

[4] F. Alebebisat, T. E. Alshabatat, and A. Zaid, "Review of Literature on Software Quality," *World Comput. Sci. Inf. Technol. J.*, vol. 8, no. 3, pp. 876–884, 2018, doi: 10.1016/j.desal.2009.09.143.

[5] L. Luo, "Software Testing Techniques," Pittsburgh, 2008. doi: 10.1201/9781439834367.axg.

[6] E. Enoiu, D. Sundmark, A. Causevic, and P. Pettersson, "A Comparative Study of Manual and Automated Testing for Industrial Control Software," *Proc. - 10th IEEE Int. Conf. Softw. Testing, Verif. Validation, ICST 2017*, no. November 2018, pp. 412–417, 2017, doi: 10.1109/ICST.2017.44.

[7] J. Mushtaq, "Different Requirements Gathering Techniques and Issues," *Int. J. Sci. Eng. Res.*, vol. 7, no. 9, pp. 835–840, 2016.

[8] M. Yadav, S. Kumar, and K. Kumar, "Quality standards for a software industry –A review," *IOSR J. Comput. Eng.*, vol. 16, no. 2, pp. 87–94, 2014, doi: 10.9790/0661-16268794.

[9] H. Sun, "Knowledge for software quality control and measurement," *Proc. 2011 Int. Conf. Bus. Comput. Glob. Informatiz.*, vol. 123, pp. 468–470, 2011, doi: 10.1109/BCGIn.2011.123.

[10] Y. Gupta, "Software Quality Assurance," *Int. J. Qual. Reliab. Manag.*, vol. 6, no. 4, pp. 56–67, 2009, doi: 10.1108/02656718910134412.

[11] P. Hegedũs, "Advances in software product quality measurement and its applications in

software evolution," Ph.D. dissertation, University of Szeged, 2015.

[12] F. S. Butt, S. Shaukat, M. W. Nisar, E. U. Munir, M. Waseem, and K. Ayyub, "Software quality assurance in software projects: A study of Pakistan," *Res. J. Appl. Sci. Eng. Technol.*, vol. 5, no. 18, pp. 4568–4575, 2013, doi: 10.19026/rjaset.5.4376.

[13] I. Sommerville, *Software Engineering*, 9th ed. Boston: Pearson Education Inc. publishing as Addison-Wesley, 2013.

[14] N. Seth and R. Khare, "ACI (Automated Continuous Integration) Using Jenkins: Key for successful embedded Software development," *2015 2nd Int. Conf. Recent Adv. Eng. Comput. Sci.*, no. 12, pp. 1–6, 2015, doi: 10.1109/RAECS.2015.7453279.

[15] S. Hossain, "Challenges of Software Quality Assurance and Testing," *Int. J. Softw. Eng. Comput. Syst.*, vol. 4, no. 1, pp. 133–144, 2018, doi: 10.15282/ijsecs.4.1.2018.11.0044.

[16] D. Young, "Software Testing : Overview," *White paper*, 2015. https://www.researchgate.net/publication/273319104_Software_Testing_Overview (accessed Jul. 15, 2020).

[17] F. Salger, S. Sauer, and G. Engels, "An integrated quality assurance framework for specifying business information systems," *Proc. CAiSE Forum 2009*, vol. 453, pp. 25–30, 2009.

[18] M. Kevitt, "Best Software Test & Quality Assurance Practices in the project Life-cycle," Ph.D. dissertation, Dublin City University, 2008.

[19] D. Galin, *Software Quality Assurance: From Theory to Implementation*, 1st ed. Harlow: Pearson Education Ltd, 2004.

[20] D. M. Owens and D. Khazanchi, "Software Quality Assurance," *Handb. Res. Technol. Proj. Manag. Planning, Oper.*, vol. 16, pp. 245–263, 2009, doi: 10.1108/02656718910134412.

[21] Perforce Software Inc, "Which Software Quality Metrics Matter ?," *White Paper*. https://www.perforce.com/resources/qac/which-software-quality-metrics-matter (accessed Jul. 15, 2020).

[22] S. A. I. B. S. Arachchi and I. Perera, "Continuous integration and continuous delivery pipeline automation for agile software project management," in *MERCon 2018 - 4th*

International Multidisciplinary Moratuwa Engineering Research Conference, 2018, pp. 156–161, doi: 10.1109/MERCon.2018.8421965.

[23] R. Torkar, “Towards automated software testing,” Ph.D. dissertation, Blekinge Institute of Technology, 2006.

[24] J. Singh and N. B. Kassie, “User’s Perspective of Software Quality,” *Proc. 2nd Int. Conf. Electron. Commun. Aerosp. Technol. ICECA 2018*, vol. 42487, pp. 1958–1963, 2018, doi: 10.1109/ICECA.2018.8474755.

[25] M. Tuteja and G. Dubey, “A Research Study on Importance of Testing and Quality Assurance in Software Development Life Cycle (SDLC) Models,” *Int. J. Soft Comput. Eng.*, vol. 2, no. 3, pp. 251–257, 2012.

[26] L. Lazic, “Managing software quality with defects,” *Telecommun. forum TELFOR2005*, vol. 13, 2005, doi: 10.1109/9780470049167.ch10.

[27] P. Williams, “Future Trends and Development Methods in Software Quality Assurance,” Master’s Thesis, Haaga-Helia University of Applied Sciences, 2017.

[28] O. Lysne, “Software Quality and Quality Management,” *Simula SpringerBriefs Comput.*, vol. 4, pp. 87–98, 2018, doi: 10.1007/978-3-319-74950-1_10.

[29] M. Shahin, M. Ali Babar, and L. Zhu, “Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices,” *IEEE Access*, vol. 5, pp. 1–32, 2017, doi: 10.1109/ACCESS.2017.2685629.

[30] T. Paulduro, “Software development quality assurance process in regional e–commerce company,” Master’s thesis, Masarik University, 2018.

[31] V. N. Maurya and R. Kumar, “Analytical Study on Manual vs Automated Testing Using with Simplistic Cost Model,” *Int. J. Electron. Electr. Eng.*, vol. 2, no. 1, pp. 23–35, 2012.

[32] M. Soni, “End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry, Continuous Integration, Continuous Testing, and Continuous Delivery,” *Proc. - 2015 IEEE Int. Conf. Cloud Comput. Emerg. Mark.*, pp. 85–89, 2016, doi: 10.1109/CCEM.2015.29.

[33] E. Chipunza, “Quality Management Challenges in Iterative Software Product Development of a Selected Software Development,” Master’s thesis, Cape Peninsula

University of Technology, 2018.

[34] SoftwareTestingHelp, “What is Software Quality Assurance (SQA): A guide For Beginners,” 2020. .

[35] L. H. Rosenberg and S. B. Sheppard, “Metrics in software process assessment, quality assurance and risk assessment,” *Int. Softw. Metrics Symp.*, pp. 10–16, 1994.

[36] P. Mahajan, “Different Types of Testing in Software Testing,” *Int. Res. J. Eng. Technol.*, vol. 03, no. 04, pp. 1661–1664, 2016.

[37] A. Kapoor, “The Definitive Guide to Agile Testing,” 2020. https://www.netsolutions.com/insights/agile-testing-for-software-products/ (accessed Jul. 20, 2020).

[38] N. Djordjevic, “Software quality standards,” *Vojnoteh. Glas. Millitary Tech. Cour.*, vol. 65, no. 1, pp. 102–124, 2017, doi: 10.5937/vojtehg65-10668.

[39] N. G. Mohammadi *et al.*, “An analysis of Software Quality Attributes and their contribution to trustworthiness,” *CLOSER 2013 - Proc. 3rd Int. Conf. Cloud Comput. Serv. Sci.*, no. December 2017, pp. 542–552, 2013, doi: 10.5220/0004502705420552.

[40] H. Hoodat and H. Rashidi, “Classification and analysis of risks in software engineering,” *World Acad. Sci. Eng. Technol.*, vol. 56, no. 8, pp. 2044–2050, 2009.

[41] N. B. J. Gamage, “The risk factors affecting to the software quality failures in Sri Lankan Software industry,” Master’s thesis, University of Sri Jayewardenepura, 2017.

[42] I. Karac and B. Turhan, “What Do We (Really) Know about Test-Driven Development?,” *IEEE Softw.*, vol. 35, no. 4, pp. 81–85, 2018, doi: 10.1109/MS.2018.2801554.

[43] D. Steidl, F. Deissenboeck, M. Poehlmann, R. Heinke, and B. Uhink-Mergenthaler, “Continuous software quality control in practice,” *Proc. - 30th Int. Conf. Softw. Maint. Evol. ICSME 2014*, vol. 30, pp. 561–564, 2014, doi: 10.1109/ICSME.2014.95.

[44] G. Mujtaba, T. Mahmood, and Z. Nasir, “A holistic approach to software defect analysis and management,” *Aust. J. Basic Appl. Sci.*, vol. 5, no. 6, pp. 1632–1640, 2011.

[45] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, “Quality and productivity outcomes relating to continuous integration in GitHub,” *Jt. Meet. Eur. Softw. Eng. Conf.*

ACM SIGSOFT Symp. Found. Softw. Eng. ESEC/FSE 2015 - Proc., vol. 10, pp. 805–816, 2015, doi: 10.1145/2786805.2786850.

[46] O. Jokinen, "Software development using DevOps tools and CD pipelines, A case study," Master's thesis, University of Helsinki, 2020.

[47] C. Chandrasekara and P. Herath, *Hands-on Azure Pipelines Understanding Continuous Integration and Deployment in Azure DevOps*, 1st ed. Apress, 2020.

[48] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "DevOps," *Ieee Comput. Soc.*, pp. 94–100, 2016.

[49] R. Jabbari, N. Bin Ali, K. Petersen, and B. Tanveer, "What is DevOps? A systematic mapping study on definitions and practices," *ACM Int. Conf. Proceeding Ser.*, vol. 201, no. March 2018, 2016, doi: 10.1145/2962695.2962707.

[50] V. Blomberg, "Turun kauppakorkeakoulu Turku School of Economics," Master's thesis, University of Turku, 2015.

[51] M. Senapathi, J. Buchan, and H. Osman, "DevOps Capabilities, Practices, and Challenges: Insights from a Case Study," pp. 1–11, 2019.

[52] A. Khan, "Practicing Continuous Integration and Continuous Delivery on AWS Accelerating Software Delivery with DevOps," 2017, doi: 10.1007/978-1-4842-4828-7_2.

[53] J. Roche, "Adopting DevOps Practices in Quality Assurance - Merging the art and science of software development," *Commun. ACM*, vol. 9, pp. 1–8, 2013, doi: 10.1145/2524713.2524721.

[54] P.-G. Stenberg, "Container-based Continuous Delivery for Clusters," Master's thesis, Lund Universtiy, 2016.

[55] C. Anderson, "Docker," *IEEE Software*, vol. 32, no. 3, IEEE, pp. 102–105, 2015.

[56] L. Hukkanen, "Adopting Continuous Integration – A Case Study," Master's thesis, Aalto University, 2015.

[57] J. Liang, S. Elbaum, and G. Rothermel, "Redefining Prioritization: Continuous Prioritization for Continuous Integration," in *International Conference on Software Engineering*, 2018, vol. 40, pp. 688–697.

[58] I. K. Moutsatsos *et al.*, "Jenkins-CI, an open-source continuous integration system, as a scientific data and image-processing platform," *J. Biomol. Screen.*, vol. 22, no. 3, 2016, doi: 10.1177/1087057116679993.

[59] T. Paulin, "DevOps in Finland : study of practitioners' perception," Master's thesis, University of Oulu, 2018.

[60] E. Koppel, "Software Test Management Tool Evaluation Framework," Master's thesis, University of Tartu, 2012.

[61] A. Mili and F. Tchier, *Software Testing Concepts and Operations*, 1st ed. New Jersey: John Wiley & Sons, Inc, 2015.

[62] AltexSoft, "Quality Assurance, Quality Control and Testing - the Basics of Software Quality Management," *White paper*, 2018. https://www.altexsoft.com/whitepapers/quality-assurance-quality-control-and-testing-the-basics-of-software-quality-management/ (accessed Jul. 10, 2020).

[63] A. Lawanna, "The Theory of Software Testing," *Intelligent Transportation Systems Journal - June 2012*, vol. 16. pp. 35–40, 2012.

[64] IEEE Standards Association, "IEEE Standard Adoption of ISO / IEC 15026-4 — Systems and Software Engineering — Systems and Software Assurance — Part 4 : Assurance in the Life Cycle IEEE Computer Society," *IEEE Comput. Soc. Spons.*, 2013.

[65] J. Ivanko, "A proposal of regression testing process for a small organization developing COST," Diploma thesis, Masaryk University, 2011.

[66] B. D. Dimitrov, V. Goyal, J. Nehenbe, and V. Papataxiarhis, "Different forms of Testing Techniques," *Int. J. Comput. Sci. Issues*, vol. 7, no. 3, pp. 11–26, 2010.

[67] N. Vaidya, "Software Testing Tutorial – Know How to Perform Testing," 2020. https://www.edureka.co/blog/software-testing-tutorial/ (accessed Jul. 20, 2020).

[68] R. Mafi, P. Mafi, and M. Malahias, "The Use of Robotic Assisted Surgery; the Current and Future Challenges," *Open Med. J.*, vol. 3, no. 1, p. 300, 2016, doi: 10.2174/1874220301603010300.

[69] O. Y. Sowunmi, S. Misra, L. Fernandez-Sanz, B. Crawford, and R. Soto, "An empirical evaluation of software quality assurance practices and challenges in a developing

country: A comparison of Nigeria and Turkey," *Springer Plus*, vol. 5, no. 1, pp. 1–13, 2016, doi: 10.1186/s40064-016-3575-5.

[70] T. Khalane, "Software Quality Assurance in Scrum," Master's thesis, University of Cape Town, 2013.

[71] C. Y. Laporte and A. April, "Software Quality Assurance in an Undergraduate Software Engineering Program," *Proc. Can. Eng. Educ. Assoc.*, vol. 13, no. 109, pp. 1–6, 2013, doi: 10.24908/pceea.v0i0.4797.

[72] I. Karamitsos, S. Albarhami, and C. Apostolopoulos, "Applying devops practices of continuous automation for machine learning," *Inf.*, vol. 11, no. 7, pp. 1–15, 2020, doi: 10.3390/info11070363.

[73] L. Leite and S. Paulo, "A Survey of DevOps Concepts and Challenges," *ACM Comput. Surv.*, vol. 52, no. 6, pp. 1–35, 2019.

[74] V. G. da Silva, M. Kirikova, and G. Alksnis, "Containers for Virtualization: An Overview," *Appl. Comput. Syst.*, vol. 23, no. 1, pp. 21–27, 2018, doi: 10.2478/acss-2018-0003.

[75] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud Container Technologies: a State-of-the-Art Review," *IEEE Trans. Cloud Comput.*, vol. 1, no. May, 2017, doi: 10.1109/TCC.2017.2702586.

[76] S. Datko, "Automate your life with Gitlab-CI," no. November, pp. 0–29, 2016.

[77] J. D. Blischak, E. R. Davenport, and G. Wilson, "A Quick Introduction to Version Control with Git and GitHub," *PLoS Comput. Biol.*, vol. 12, no. 1, 2016, doi: 10.1371/journal.pcbi.1004668.

[78] Y. Perez-Riverol *et al.*, "Ten Simple Rules for Taking Advantage of Git and GitHub," *PLoS Comput. Biol.*, vol. 12, no. 7, pp. 1–11, 2016, doi: 10.1371/journal.pcbi.1004947.

APPENDIX A – Event logs

System event logs from January to October

January

Event Logs

Filters

Groups

DefaultGroup

From

2020-01-01 00:00:00

To

2020-01-31 00:00:00

Filter

Detailed Event Information 0 | Event Escalation Overview | Event Log Analysis | BI Analytics

Application	Error	Detail	Warning	Debug	Info	CriticalError	None
ASTIR_HQ	236	10578	10	8935	30	157	0
ASTIR_Translator	132	578	19	367	8	78	0
ASTIR_EventLogServic	108	15	25	21	84	69	0
ASTIR_Extractor	213	406	101	3096	10	95	0
ASTIR_Communicator	785	307	203	299	208	409	0

February

Event Logs

Filters

Groups

DefaultGroup

From

2020-02-01 00:00:00

To

2020-02-29 00:00:00

Filter

Detailed Event Information 0 | Event Escalation Overview | Event Log Analysis | BI Analytics

Application	Error	Detail	Warning	Debug	Info	CriticalError	None
ASTIR_HQ	223	389	8	480	10	147	0
ASTIR_Translator	127	6	20	4053	8	71	0
ASTIR_EventLogServic	97	10	25	5	30	64	0
ASTIR_Extractor	185	6795	102	951	472	96	0
ASTIR_Communicator	764	480	198	299	205	369	0

March

Event Logs

Filters

Groups

DefaultGroup

From

2020-03-01 00:00:00

To

2020-03-31 00:00:00

Filter

Detailed Event Information 0 | Event Escalation Overview | Event Log Analysis | BI Analytics

Application	Error	Detail	Warning	Debug	Info	CriticalError	None
ASTIR_HQ	222	7856	9	6952	19	131	0
ASTIR_Translator	118	326	18	16	10	71	0
ASTIR_EventLogServic	98	12	24	41	63	61	0
ASTIR_Extractor	179	183	114	1678	13	91	0
ASTIR_Communicator	648	142	201	98	159	301	0

April

Event Logs

Filters

Groups

DefaultGroup

From

2020-04-01 00:00:00

To

2020-04-30 00:00:00

Filter

Detailed Event Information 0 | Event Escalation Overview | Event Log Analysis | BI Analytics

Application	Error	Detail	Warning	Debug	Info	CriticalError	None
ASTIR_HQ	214	5241	10	4586	17	125	0
ASTIR_Translator	113	265	18	14	12	69	0
ASTIR_EventLogServic	85	11	26	34	65	58	0
ASTIR_Extractor	171	164	118	1243	13	89	0
ASTIR_Communicator	612	98	215	85	126	289	0

May

Application	Error	Detail	Warning	Debug	Info	CriticalError	None
ASTIR_HQ	209	3152	9	3012	14	121	0
ASTIR_Translator	109	123	19	12	10	67	0
ASTIR_EventLogServic	81	15	27	36	53	47	0
ASTIR_Extractor	165	156	120	953	11	86	0
ASTIR_Communicator	589	75	219	71	114	281	0

June

Application	Error	Detail	Warning	Debug	Info	CriticalError	None
ASTIR_HQ	194	2415	10	2541	0	119	0
ASTIR_Translator	101	114	21	16	23	61	0
ASTIR_EventLogServic	78	12	28	40	15	49	0
ASTIR_Extractor	164	136	125	532	15	82	0
ASTIR_Communicator	584	52	198	58	102	273	0

July

Event Logs

Filters

Groups

DefaultGroup

From

2020-07-01

To

2020-07-31

Filter

Detailed Event Information 0 | Event Escalation Overview | Event Log Analysis | BI Analytics

Application	Error	Detail	Warning	Debug	Info	CriticalError	None
ASTIR_HQ	186	1623	10	1745	13	112	0
ASTIR_Translator	92	101	21	7	19	59	0
ASTIR_EventLogServic	76	15	28	35	16	42	0
ASTIR_Extractor	164	152	128	312	11	71	0
ASTIR_Communicator	502	41	201	41	94	271	0

August

Event Logs

Filters

Groups

DefaultGroup

From

2020-08-01

To

2020-08-31

Filter

Detailed Event Information 0 | Event Escalation Overview | Event Log Analysis | BI Analytics

Application	Error	Detail	Warning	Debug	Info	CriticalError	None
ASTIR_HQ	171	956	11	1236	15	105	0
ASTIR_Translator	92	85	22	12	17	54	0
ASTIR_EventLogServic	69	7	28	26	14	42	0
ASTIR_Extractor	158	124	129	196	9	69	0
ASTIR_Communicator	489	35	210	32	75	269	0

September

Event Logs

Filters

Groups

DefaultGroup

From

2020-09-01

To

2020-09-30

Filter

Detailed Event Information 0 | Event Escalation Overview | Event Log Analysis | BI Analytics

Application	Error	Detail	Warning	Debug	Info	CriticalError	None
ASTIR_HQ	152	645	11	984	16	101	0
ASTIR_Translator	89	91	22	16	13	49	0
ASTIR_EventLogServic	70	16	28	12	18	40	0
ASTIR_Extractor	141	101	132	85	15	63	0
ASTIR_Communicator	481	26	211	41	72	264	0

October

Event Logs

Filters

Groups

DefaultGroup

From

2020-10-01

To

2020-10-31

Filter

Detailed Event Information 0 | Event Escalation Overview | Event Log Analysis | BI Analytics

Application	Error	Detail	Warning	Debug	Info	CriticalError	None
ASTIR_HQ	143	312	12	789	21	98	0
ASTIR_Translator	82	65	23	36	14	45	0
ASTIR_EventLogServic	65	12	30	11	16	39	0
ASTIR_Extractor	126	86	134	69	12	57	0
ASTIR_Communicator	469	14	215	23	51	256	0

APPENDIX B – Code coverage

Below details the test results of January, that is before the results were gathered, to November when all the results were collected. The graphic below reports January's numbers.

Code Coverage Results

saboozoo_SABOOZOO_2020-01-10.09_09_52

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
saboozoo_SABOOZOO_2020-01-10.09_09_52.coverage	55205	99.91%	49	0.09%
devlib.datastore.dll	19774	100.00%	0	0.00%
DevLib.DataStore.Central	159	100.00%	0	0.00%
DevLib.DataStore.DAL	129	100.00%	0	0.00%
DevLib.DataStore.DAL.Astir	4164	100.00%	0	0.00%
DevLib.DataStore.DAL.Astir.Helpers	363	100.00%	0	0.00%
DevLib.DataStore.DAL.Custom	52	100.00%	0	0.00%
DevLib.DataStore.DAL.MTB_Database	10240	100.00%	0	0.00%
DevLib.DataStore.DAL.MobileToolbox	87	100.00%	0	0.00%
DevLib.DataStore.Library	692	100.00%	0	0.00%
DevLib.DataStore.Models.ActionTracker	375	100.00%	0	0.00%
DevLib.DataStore.Models.Astir	539	100.00%	0	0.00%
DevLib.DataStore.Models.Astir.Custom	136	100.00%	0	0.00%
DevLib.DataStore.Models.Astir.EditReports	223	100.00%	0	0.00%
DevLib.DataStore.Models.Astir.Partial	4	100.00%	0	0.00%
DevLib.DataStore.Models.Custom	92	100.00%	0	0.00%
DevLib.DataStore.Models.MTB_Database	1027	100.00%	0	0.00%
DevLib.DataStore.Models.MobileToolbox	19	100.00%	0	0.00%
DevLib.DataStore.Models.Platform	10	100.00%	0	0.00%
DevLib.DataStore.ReadDAL	392	100.00%	0	0.00%
DevLib.DataStore.ReadDAL.Astir	628	100.00%	0	0.00%
DevLib.DataStore.ReadDAL.MTB	31	100.00%	0	0.00%
DevLib.DataStore.ReadModels.Astir	276	100.00%	0	0.00%
DevLib.DataStore.ReadModels.MTB	22	100.00%	0	0.00%
DevLib.DataStore.ReadModels.MTB_Database	75	100.00%	0	0.00%
DevLib.DataStore.ViewModels.Custom	39	100.00%	0	0.00%
devlib.datastore.tests.dll	22441	100.00%	1	0.00%
DevLib.DataStore.Tests	188	99.47%	1	0.53%
DevLib.DataStore.Tests.ASTIR	10815	100.00%	0	0.00%
DevLib.DataStore.Tests.Astir.DatabaseInteractions	266	100.00%	0	0.00%
DevLib.DataStore.Tests.MTB_Database	9212	100.00%	0	0.00%
DevLib.DataStore.Tests.MTB_Database.DatabaseInteractions	1748	100.00%	0	0.00%
DevLib.DataStore.Tests.MobileToolbox	123	100.00%	0	0.00%
WebLib.Tests.MTB_Database	81	100.00%	0	0.00%
WebLib.Tests.MTB_Database.DatabaseInteractions	8	100.00%	0	0.00%
devlib.logger.dll	489	100.00%	0	0.00%
devlib.testing.dll	758	94.04%	48	5.96%
DevLib.Testing	48	100.00%	0	0.00%
DevLib.Testing.Asserts	422	100.00%	0	0.00%
DevLib.Testing.Containers	26	61.90%	16	38.10%
DevLib.Testing.Helpers	262	89.12%	32	10.88%

The figure below reports November's numbers.

Code Coverage Results

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
saboozoo_SABOOZOO_2020-11-10.14_58_21.coverage	21094	41.78%	29390	58.22%
devlib.datastore.dll	7083	40.35%	10469	59.65%
DevLib.DataStore.Central	17	10.69%	142	89.31%
DevLib.DataStore.DAL	81	62.79%	48	37.21%
DevLib.DataStore.DAL.Astir	486	11.52%	3733	88.48%
DevLib.DataStore.DAL.Astir.Helpers	73	20.11%	290	79.89%
DevLib.DataStore.DAL.Custom	52	100.00%	0	0.00%
DevLib.DataStore.DAL.MTB_Database	4649	56.61%	3563	43.39%
DevLib.DataStore.DAL.MobileToolbox	8	9.20%	79	90.80%
DevLib.DataStore.Library	193	28.01%	496	71.99%
DevLib.DataStore.Models.ActionTracker	204	54.69%	169	45.31%
DevLib.DataStore.Models.Astir	94	17.38%	447	82.62%
DevLib.DataStore.Models.Astir.Custom	194	93.27%	14	6.73%
DevLib.DataStore.Models.Astir.EditReports	163	100.00%	0	0.00%
DevLib.DataStore.Models.Astir.Partial	0	0.00%	4	100.00%
DevLib.DataStore.Models.Custom	92	100.00%	0	0.00%
DevLib.DataStore.Models.MTB_Database	491	57.29%	366	42.71%
DevLib.DataStore.Models.MobileToolbox	0	0.00%	19	100.00%
DevLib.DataStore.Models.Platform	10	100.00%	0	0.00%
DevLib.DataStore.ReadDAL	105	26.79%	287	73.21%
DevLib.DataStore.ReadDAL.Astir	31	4.94%	597	95.06%
DevLib.DataStore.ReadDAL.MTB	0	0.00%	31	100.00%
DevLib.DataStore.ReadModels.Astir	97	36.88%	166	63.12%
DevLib.DataStore.ReadModels.MTB	4	18.18%	18	81.82%
DevLib.DataStore.ViewModels.Custom	39	100.00%	0	0.00%
devlib.datastore.tests.dll	1368	6.93%	18375	93.07%
DevLib.DataStore.Tests	0	0.00%	189	100.00%
DevLib.DataStore.Tests.ASTIR	3	0.03%	10871	99.97%
DevLib.DataStore.Tests.Astir.DatabaseInteractions	74	27.82%	192	72.18%
DevLib.DataStore.Tests.MTB_Database	0	0.00%	6393	100.00%
DevLib.DataStore.Tests.MTB_Database.DatabaseInteractions	1291	71.40%	517	28.60%
DevLib.DataStore.Tests.MobileToolbox	0	0.00%	124	100.00%
WebLib.Tests.MTB_Database	0	0.00%	81	100.00%
WebLib.Tests.MTB_Database.DatabaseInteractions	0	0.00%	8	100.00%
devlib.logger.dll	488	100.00%	0	0.00%
devlib.testing.dll	299	37.10%	507	62.90%
DevLib.Testing	8	16.67%	40	83.33%
DevLib.Testing.Asserts	231	54.74%	191	45.26%
DevLib.Testing.Containers	15	35.71%	27	64.29%
DevLib.Testing.Helpers	45	15.31%	249	84.69%

APPENDIX C – Coding activity

The following presents activity insights, which the version management and collaboration platform provided.

Pulse
Contributors
Commits
Code frequency
Dependency graph
Network
Forks

Jan 7, 2018 – Nov 22, 2020

Contributions: **Commits**

Contributions to master, excluding merge commits