A Quality Assurance Model for Airborne Safety-Critical Software

Habib A. ElSabbagh

A Thesis in the

Department of Mechanical and Industrial Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science (Mechanical Engineering) at

Concordia University

Montreal, Quebec, Canada

April 2006

Your file Votre référence
ISBN: 0-494-14305-3
Our file Notre référence
ISBN: 0-494-14305-3

ABSTRACT

A Quality Assurance Model for Airborne Safety-Critical Software

Habib A. ElSabbagh

Software applications in which failure may result in possible catastrophic consequences on human life are classified as safety-critical. These applications are widely used in a variety of fields and systems such as airborne systems, nuclear reactors' control, and medical diagnostic equipment. Unfortunately, the world has seen several accidents and tragedies caused by software failure error or where such failure/error was part of the problem. This thesis looks into safety-critical software embedded in airborne systems. It proposes a lifecycle specially modeled for the development of safety-critical software in aerospace and in compliance with the DO-178B standard and a software quality assurance (SQA) model based on a set of four acceptance criteria that builds quality into safety-critical software throughout its development. The thesis also provides frameworks and guidelines for the implementation of the proposed SQA model in addition to sets of rules defining how to assess the software development with respect to the four acceptance criteria.

In memory of

Prof. Jaroslav Svoboda

ACKNOWLEGMENTS

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. I am deeply indebted to my supervisor Dr. Nadia Bhuiyan whose help, stimulating suggestions, and encouragement helped me in all the time of research for and writing of this thesis. Her support and understanding were vital for the completion of this work. I would like to thank CMC Electronics and the Consortium for Research and Innovation in Aerospace in Quebec (CRIAQ) for giving me the opportunity to work on a project that was vital for this thesis. I also extend my thanks to Dr. Henry Hong and M. Jian Chen of Concordia University for their help and guidance with the project.

This thesis has been a long journey. A journey in which I was supported by a wonderful family. I sincerely thank my parents, my sisters, and my uncle Fares without whom this would not have been possible. Finally, I would like to extend my sincere gratitude and appreciation to all my friends who were near me when I needed them, with special thanks to M. Antoine Khalil.

To

my parents Marie and Assaad,

and my uncle Fares

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 INTRODUCTION

Embedded systems are special-purpose computer systems, which are encapsulated or mounted into the device they are built to control. Embedded systems play a critical role in human beings' daily life. They are present in a wide range of applications from personal systems to large industrial ones. In modern embedded systems, there is always a software element. Furthermore, that software is typically used to control a larger system of which it is only one part.

A safety critical system is a system where a malfunction can result in loss of life, injury or illness, or serious environmental damage. The emphasis of this thesis is on the software element of safety critical systems, which for convenience is often referred to as safety critical software.

On the ground, software in medical systems may be directly responsible for human life, such as metering safe amounts of X-rays. The accidents caused by Therac-25, an accelerator used in medical radiation therapy, are a very good example of the harm resulting from poorly designed software; "between June 1985 and January 1987, six known accidents involved massive overdoses by the Therac-25 with resultant deaths and serious injuries" (Leveson et al., 1993) at treatment centers around the U.S. and in Canada. Within cars, a failure of software involved in functions such as engine management, anti-lock brakes, and traction control, could potentially increase the likelihood of a road accident. In nuclear power plants, reactors' control systems are highly dependent on their software component. Railway signalling depends on software based systems that must enable controllers to direct trains and prevent them from colliding.

In the aerospace industry, an example of a safety critical system where software is a major component is aircrafts' fly-by-wire control system, where the pilot inputs commands to the control computer using a joystick, and the computer manipulates the actual aircraft controls. Fly-by-wire has become an industry standard for Airbus and for every new Boeing design since the Boeing 777. The lives of hundreds of passengers are totally dependent upon the continued correct operation of such a system.

Currently the trend in the use of safety-critical software in the aerospace industry is mostly concentrated on avionic systems. Whether used for flight navigation or approach (Navaids, Instrument Landing System), aircraft control (Autopilot), flight management and planning (FMS) or cockpit electronic displays (Glass Cockpit), these systems have to work in "real time" while maintaining high standards of safety and reliability.

Unfortunately the world has seen aircraft fatal accidents caused by software failure error or where such failure/error was part of the problem: a study of the 1995 American Airline crash in Cali, Columbia blamed part of the problem on the FMS' software as it presented insufficient and conflicting information to the pilots, who got lost (Airclaims Ltd, 2005). In 1998, a China Airlines aircraft crashed during an attempted go-around following an ILS approach to Taipei (Airclaims Ltd, 2005). The radar that could have prevented the 1997 Korean Air crash into the side of Nimitz Hill in Guam was hobbled by software problem. (Airclaims Ltd, 2005).

In areas which have been traditionally concerned with safety critical systems, such as the aviation industry, embedded control systems and their software components have to be certified by government authorities and certification bodies such as Transport Canada, the Federal Aviation Administration (FAA), and the Joint Aviation Authorities (JAA).

1.1 Purpose, Objectives, and Scope

The tragic nature of safety-critical software failure's consequences makes high quality and extreme reliability essential requirements in such types of software. While standards for certification and development of safety-critical software have been developed by authorities and the industry, very little research has been done addressing safety-critical software quality.

The research behind this thesis focuses on safety-critical software quality. In general, it looks into the following three questions:

- Is safety-critical software quality controllable and manageable like any other software?
- What flow should all safety-critical software development activities follow?
- How to build and manage safety-critical software quality?

After studying safety-critical software quality and demonstrating how it differs from other software and products, the thesis proposes a lifecycle model for all safety-critical software development activities. It then proposes a new software quality assurance model specially designed for the development of such software products. The proposed model advocates building quality into the product right from the moment the development project is initiated; it provides frameworks on how to monitor a product's quality throughout the development. The objective is to produce a high quality, reliable, and certifiable safety-critical software on time, within budget, and through a development lifecycle free of iterations that are common in software development.

The proposed work in this thesis mainly targets safety-critical software projects developed for aerospace applications. Furthermore, the main interest is "make-to-order" projects which have more specific requirements and operational functionalities in comparison to projects for commercial off the shelf products.

1.2 Methodology

Literature on previous work done in the field of airborne safety-critical software was meticulously investigated and studied. Due to limited research and literature available, the study was expanded to safety-critical software research in other fields, mainly nuclear reactor's control systems. In addition, software quality standards and guidelines were reviewed and the standard DO-178B, which is the de facto standard in the aerospace industry, is at the center of interest. The software quality assurance proposed in this thesis is directly related to and complementary to the DO-178B. The DO-178B is reviewed in Chapter 3.

The proposed work in this thesis is also the result of the experience and lessons learned while working on software quality assurance and certification issues of a collaborative project entitled Dynamic Test Bed (DTB) for Flight Management System

(FMS). The project is a joint research and development project between CMC Electronics and Concordia University under the auspices of the Consortium for Research and Innovation in Aerospace Quebec (CRIAQ). The project deals with the development of a real-time dynamic test bed (DTB) which will serve as a comprehensive design/test tool for the flight management system (FMS) product line of CMC Electronics. The DTB will facilitate more efficient and more economic design of FMS systems.

The DTB is used to simulate the FMS with a real aircraft environment and conditions. The main objective is to reduce the number of flight tests by getting flight test credit by Transport Canada, prepare and run the entire engineering and Transport Canada flight-test, and reproduce customer problems with the same conditions under the DTB. The authority can do an IFR flight test under the DTB. Systems tests can be performed under the DTB like evaluation and test of the dynamic performance of the CMA-900 Flight Management System (FMS); act as a Vertical Navigation development test bed; and demonstrate the operation of all FMS modes for customers and other interested parties.

While a failure of the DTB will not result in any human tragedy or injury, the software components of the DTB are considered to be safety-critical since they simulate real-time flight scenarios used to test FMS's. A failure in a flight scenario might not detect an error in an FMS which once installed on an airplane might fail and result in terrible consequences.

1.3 Organization

This introductory chapter is followed by five other chapters. Chapter 2 provides a literature review. It starts by reviewing research done on quality and software quality in general, and then it provides a review of safety-critical software quality literature.

Chapter 3 explores software quality and how quality control, methods, and tools can be applied to safety-critical projects. Through an analysis, it is proven in this chapter that testing for errors and sampling safety-critical software is not feasible. The chapter then reviews software quality assurance and its benefits and how it can benefit safety-critical software projects. The last part of Chapter 3 outlines some of the standards used in software development and particularly DO-178B which is used for airborne software development.

Chapter 4 starts with a review of several lifecycles modeled for software development activities; it then proposes a lifecycle modeled by the author for the particular case of safety-critical software development projects.

Chapter 5 explains the software quality assurance model proposed in this thesis. In the first part of the chapter, four criteria at the center of the proposed model and their benefits are explained. The second part provides frameworks and guidelines for each of the four criteria and demonstrates how the proposed model should be applied concurrently to all development activities throughout the lifecycle.

Chapter 6 concludes this thesis with a summary outlining all the major points proposed and explaining their benefits and advantages. Furthermore, this chapter provides an outlook at future work and research that can follow this thesis.

2.0 LITERATURE REVIEW

Quality has always been embedded in production activities. A high quality product was the pride of ancient craftsmen, as well as the pride of villages and cities that specialized in a certain production activity.

During the Middle Ages, quality was to a large extent controlled by the long periods of training required by the guilds. The Industrial Revolution introduced the concept of labour specialization which brought a decline in workmanship. In the early periods, quality was not affected much because products produced were not complicated. As products became more complicated and jobs more specialized, it became necessary to inspect products after manufacture. Today's customers have very high expectations regarding quality and reliability of products and expect the highest of standards from all organizations within the economy. This includes manufacturing, services, and software sectors.

This chapter is a literature review of studies made on software quality as well as models and standards developed for software production and safety-critical software in particular. The chapter starts with a review of several studies and theories showing the evolution of quality as a scientific concept, and then it reviews the growth of software quality studies. The final part of this chapter provides a literature review on quality and assessment of safety-critical software. Unfortunately, literature on safety-critical software quality is very limited as research in this field has only recently been getting more awareness. As noted by Kornecki et al. (2004a) producers of such software products tend to rely on industry standards and develop their own quality procedures and keep them confidential due to the characteristics of the end product and its use.

In the field of safety-critical software, several standards exist and several others are under development especially for software applications in airborne systems, nuclear reactors control, and railway signalization systems. For avionics and airborne software, the DO-178B and its European version ED-12B remain as the de facto quality standard used as guidelines in the development and certification of high-quality safety-critical software products. Chapter 3 provides a summary of those standards and a presentation of DO-178B.

2.1 General Quality Studies

The first scientific work and study to be done on quality was introduced in 1924 by Shewhart of Bell Telephone Laboratories (Shewart, 1931). Shewhart developed a statistical chart for the control of product variables; this chart is regarded as the foundation of statistical quality control. Shewhart also introduced a process improvement approach consisting of four major steps: Plan, Do, Check, and Act. His argument that "quality and productivity improve as process variability is reduced" (Shewart, 1931) was confirmed by Japanese engineers in the 1950s that triggered an extreme focus on quality by Japanese companies.

Based on the work of Shewart, Deming (1986) introduced the concept of having quality management and a quality policy for the whole organization. He argued that "it is not sufficient for everyone in the organization to be doing his best: instead, what is required is that there be a consistent purpose and direction in the organization" (O'Regan, 2002). By doing so, organizations will cut costs by saving on reworking defective products and, in parallel, increasing productivity. By the end of his career in the 1980s, Deming –whose ideas were highly influential in the Japanese organizations– proposed 14 principles of action that helped transform the western style of management of an organization to a quality and customer focused organization and repositions itself on the market by offering high-quality products. These principles include (Deming, 1986):

- Constancy of purpose
- Quality built into the product
- Continuous improvement culture
- Institution of training
- Self improvement

In parallel to Deming's work, Juran (1951) emphasized the role of management in quality assurance. He argued that management is directly responsible for quality and that it must ensure quality planning, controlling and improvement. Juran's idea on quality became widely known as the "Juran Trilogy": Plan, Control, and Improve. Juran also defined his "Breakthrough and Control" theory; an approach to achieve a new quality performance level. It consists of showing graphically the old performance level with occasional spikes in order to aim to a more consistent quality level. Juran called the difference between the old performance level and the one aimed for the "chronic disease"; the breakthrough can be achieved when this "disease" is diagnosed and cured making the old performance obsolete (Juran, 1951).

2.2 Software Quality Studies

Software quality and software industrial sectors were highly influenced by the work of Crosby. Crosby (1980) introduced the concept of "doing things right from the first time" calling it the "Zero Defects (ZD) program". Crosby rejected the notion of "Acceptable Quality Level" saying that it is a commitment to produce imperfect material and that it conditions people to believe in the inevitability of errors. According to Crosby, the two main reasons of defects are "lack of knowledge" and "lack of attention" of the individual; while the "lack of knowledge" can be measured and treated by training, the "lack of attention" is a mindset that requires a change of attitude by the individual.

Crosby introduced a 14-step quality improvement program to help organizations reach the Zero Defects level. The success of this program is highly dependent on the commitment of the management to it and requires an organization-wide quality improvement to be set up. The 14 steps include (Crosby, 1980):

- Management Commitment: Raise quality awareness throughout the organization.
- Quality Improvement Team: Set up a team of representatives of all departments to monitor improvements and actions.
- Quality Measurements: Determine the status of quality in each area.

Crosby maintained that a successful implementation of the Zero Defects program will result in "higher productivity due to less reworking of defective products" (Crosby, 1980).

In order to monitor the improvement of the organization towards the Zero Defects level, Crosby established a Quality Management Grid. Organizations are ranked according to five ascending maturity levels: “Uncertainty, Awakening, Enlightenment, Wisdom, and Certainty” (Crosby, 1980). The maturity level measurements are made taking into consideration the following: “management understanding and attitude towards quality, quality organization status, problem handling, cost of quality, quality improvement actions, and summation of company quality posture”.

Other researchers looked into software quality from a requirements evolution viewpoint. During a software development project, requirements changes become increasingly expensive and risky and each requirements change affects the success of the project as well as system characteristics (e.g., dependability, safety, reliability, usability, etc.) that are crucial for the system functionalities. As demonstrated by Haney (1972), “a change in one part of the software causes a change in another part of the software through a dependency that had not been considered by the instigator of the change”; this phenomenon is known by the “ripple effect”.

Sommerville et al. (1997) proposed guidelines for Requirements Management. Among the main guidelines for requirements management, there are three related activities: Define Change Management Policies; Identify Global System Requirements; Identify Volatile Requirements. They argued that “the definition of change management policies is really important in order to have a formal mechanism to deal with changes into requirements”. The importance of identifying Global System Requirements is based on the fact that they “are very expensive to change (e.g., architecture changes) and may involve different stakeholders”. Identifying Volatile Requirements “may help to design the system so that volatile requirements are isolated from the rest of the system”. Software quality is highly affected by recurrent requirements changes and their resulting ripple effects. Sommerville's guidelines can be used to analyse requirements evolution throughout a software project. This analysis helps to keep track of changes within the process and to assess their impact.

Hooks et al. (2001) stressed the need of measuring requirements' quality saying that "the main motivation of measuring requirements is to identify opportunity for improvement". They pointed out that most of the time this view is not recognised by managers, who "believe that measuring requirements is costly (in terms of resources) and time consuming". They introduced a method to analyse changes in requirements based on simple measurements and statistics derived from the analysis of requirements counting, trends, percentages, as well as classifications. Moreover, they represent a means for comparing projects.

Weinberg (1997) stressed how the "assumption of fixed requirements is a weak point in most of the work in Software Engineering". He viewed the software development process as a composite of two "twin processes": one to develop a requirements product and one to develop a software product. Accordingly the quality of the final product can be controlled by controlling those two processes.

2.3 Safety-critical Software Quality

Software quality has been studied thoroughly. However, and as demonstrated by Kornecki et al. (2003a), very little research has been done in the area of safety-critical software quality and quality assurance, which both "[...] have not received enough attention in the literature".

The early studies were interested in whether independently developing variants of the same software could contribute to the increase of safety. These studies dealt with the quality of safety-critical software as being synonymous with reliability; if the software is reliable then it is of good quality and hence safe. The basic idea has been to first produce independently two variants of software to solve the same task then demonstrate that each single variant has achieved certain reliability, and finally conclude the probability of common failure by multiplying the probability of failure of the first software by the probability of failure of the second (i.e. P(1&2)=P(1)*P(2)). This approach is very similar to what is known as "parallel systems", a traditional approach used in a wide range of applications (e.g. aircrafts and rockets). Its objective is to optimize through redundancy the reliability of the system: even though only one component is required at least two are built into the system so that if the first fails the system can still operate using the second. A practical example of this approach in safety-critical systems is the protection systems design modification project at the Czech nuclear power plant of Temelin, where two different computer systems (primary and diverse) were used for the fulfillment of the same functions (Kharchenko et al., 1999).

Eckhardt and Lee (1985) objected the conclusion that the common probability of failure of both developed software is the product of the probability of failure of each. They theoretically showed that this conclusion should be "augmented by a Variance (*Var*): a measure of the varying difficulty throughout the input data space, namely the variance of the probability of committing programming errors over the input domain of the two diverse program variants". Accordingly, the probability of failure of a system is equal to: $P(system) = [P(1) \times P(2) \times ... \times P(n)] + Var$. Eckhardt and Lee's work was followed by several researchers dedicated to the estimation of the variance. Such research was conducted by Littlewood and Miller (1987) and Ehrenberger and Saglietti (1993).

Littlewood and Miller (1987) looked into the requirement to develop each component independently from the other. They computed the Variance (*Var*) as the average probability of failure on an input resulting from the use of different methodologies to develop each component. They also said that the variance can be either positive or negative. According to Ehrenberger and Saglietti (1993), "if the input domain of a diverse software system is processed by *K* disjoint input channels, and if the software of the individual channels fails independently, the probability of common failures is increased by a factor close to *K*: $P(system) = P(1) \times P(2) \times ... \times P(n) \times K$ ".

The independent parallel software approach for safety-critical applications might increase the reliability of the whole system but it still requires that all the software used be of extremely high quality. For example, if the probability of failure of a system made of two equally reliable software elements is required to be at least 10^{-10} then the probability of failure of each element should be at least 10^{-5}. Therefore, a quality procedure is still required for the production of each software element.

Another approach to safety-critical software quality was trying to apply to software systems safety analysis techniques successfully used in non-computer-based systems. Fault Tree Analysis (FTA), Failure Mode and Effect Analysis (FMEA), and Hazard Operability Analysis (HAZOP) are examples of those techniques.

FTA is a graphical representation of the major faults or critical failures associated with a product, the causes for the faults, and potential countermeasures. The tool helps identify areas of concern and corrective actions. FMEA is a quality planning tool that examines potential product or process failures, evaluates risk priorities, and helps determine remedial actions to avoid identified problems. FMEA mainly investigates the following nine areas: Mode of Failure, Cause of Failure, Effect of Failure, Frequency of Occurrence, Degree of Severity, Chance of Detection, Risk Priority, Design Action, and Design Validation. HAZOP is based on a theory that assumes that risk events are caused by deviations from the design or operating intentions. It is a systematic brainstorming technique for identifying hazards. It often uses a team of people with expertise covering the design of process or product and its application.

In many cases, these techniques cannot be directly applied to software due to special characteristics of software-controlled applications like discrete nature of processes, complexity, domination of design faults, or real-time constraints.

In order to overcome this problem, some tried adapting those techniques to software and some looked into extending the techniques to software safety analysis. For instance, Leveson et al. (1991) adapted FTA to safety-critical software while Redmill et al. (1999) developed a method to adapt HAZOP with respect to computer systems. Maier (1995) introduced a "method of informal safety analysis of software-based systems using FTA and FMEA"; Maier's work has been applied in the nuclear industry. On the other hand, Cichocki and Górski (2000) studied theoretically "extending the FMEA method to make it suitable to analyze object-oriented software designs".

Elliot et al. (1994) described a "safety improvement process" which "must assure that the software safety analysis is closely integrated with the system safety analysis and the software safety is explicitly verified". They emphasized the analysis first and verification next. They described the objectives of system safety analysis as:

- "Identifying the key inputs into the software requirements specification, such as hazardous commands, limits, timing constraints, sequence of events, voting logic, failure tolerance, etc.
- Creating and identifying the specific software safety requirements in the body of the conventional software specification.
- Identifying which software design components are safety-critical".

The objectives of verification were to:

- "Apply specific verification and validation techniques to ensure appropriate implementation of the software safety requirements.
- Create test plans and procedures to satisfy the intent of the software safety verification requirements.
- Introduce any necessary corrective actions resulting from the software safety verification".

In a report for the British Computer Society, Wichmann (1999) suggested focusing the assessment of safety-critical software on the development tools used, which "normally have direct influence on the safety system, such as compilers, but also design tools that generate code for safety related target systems". Wichmann proposed the following criteria for development tools assessment:

- "Ease of validation of the tool result.
- Software techniques and processes used to develop the tool.
- Software techniques and methodologies used by the tool.
- Quality system of the tool developer.
- Previous use of the tool".

Studying the quality of safety-critical software used in nuclear reactors control systems, Bowen et al. (2003) suggested a new direction for the use of formal (i.e. mathematical) methods. A set of formal methods is a methodological framework for system specification, production and verification. The authors proposed using formal methods for establishing regulatory requirements which "could help to eliminate various understanding of problems or ambiguity of informal definitions, to allow rigorous assessment of satisfaction to requirements and finally to increase the safety level of a system". For describing regulatory requirements, they chose the formal method known as "Z notation" (named after Zermelo-Fränkel) which is usually used for modelling specifications and intended behaviour of a system.

During the FAA Software conference held in Dallas/Ft.Worth in May 2002, a survey was developed with the cooperation of the FAA and NASA Langley on safety-critical software development. Personnel from organizations such as Airbus, Boeing, Goodrich, Honeywell, Raytheon, Sikorski, and Verocel were surveyed. The following are the major observations made from the responses received (Kornecki et al., 2004a):

- "The current qualification process using DO-178B is too cumbersome, not practical and not suited to the industry needs.
- The rigor of data required for the development tool qualification, makes it impractical to qualify a COTS (Commercial, Off-the-Shelf) tool (not many COTS vendors are likely to provide extensive internal data required by the DO-178B for their product).
- 90% re-use of qualification data would be recommended way to simplify the qualification process.

- o Most of the tools involved in the qualification process are in-house products developed by the applicants using the DO -178B process.
- o Functionality was the primary criteria used for evaluation".

The problem with avionic COTS safety-critical software development was researched by Kornecki of the Riddle Aeronautical University, Florida, and Zalewski of the Florida Gulf Coast University. With their research team, they conducted a three phase research between 2002 and 2004 on assessment of development tools for COTS safety-critical software. Based on Wichmann's suggestions, they argued that "the methods and the tools used [in software design] affect the quality of the produced software and, therefore, the overall system safety". They proposed two metrics to evaluate the designed architectures. They approached software architectures by partitioning them into "event threads". They called the result of the partitioning "thread model" of a problem, defined as "a set of event threads accounting for all relevant event occurrences in the problem, [where] each event thread in the model generally becomes a software task" (Kornecki et al., 2003b).

The first metric, concurrency level, "is defined as the maximum number of events ever occurring in an arbitrarily short interval. The number of threads in an optimal model of a problem is equal to the problem's concurrency level". An optimal thread model is one where all threads co-occur. This first metric "allows building and evaluation of the optimal architecture, according to the criterion of a minimal number of threads in the problem domain" (Kornecki et al., 2004a).

The second metric, software sensitivity, "is a measure of how fast the software responses change when deadlines are increased or decreased" (Kornecki et al., 2004b).

Kornecki et al. (2004b) demonstrated theoretically the usefulness and effectiveness of their method and are still conducting some experiments; their work is concentrated on COTS safety-critical software.

The interest of this thesis is “make-to-order” safety-critical projects developed for airborne applications. Those projects have more specific requirements and operational functionalities in comparison to COTS projects and hence require a different approach in quality.

2.4 Summary

Interests in studying quality as a science started in the early 1900s and evolved as industry grew over the century. Quality became a major industrial field and an essential component in market competition for all types of products. Software products were no exception: as the use of software spread to all kinds of applications, Software Quality was studied and several concepts were introduced.

Very little research has been done on quality and quality assurance for safety-critical software projects. Most of the work was done in the field of nuclear reactors' control and COTS avionics software.

This thesis proposes a software quality assurance approach complementary to DO-178B by offering a method to assess all development lifecycle stages with respect to their corresponding DO-178B activities, actions, and requirements. This method targets mainly safety-critical software development for airborne applications.

The following chapter explains software quality, quality control and how it applies to safety-critical software products, as well as software quality assurance.

3.0 SOFTWARE QUALITY

In this thesis, a software quality assurance model is proposed for safety-critical software that ensures a high quality end product. In order to understand it better, it is important to understand what software quality, software quality control, and software quality assurance are.

This chapter is composed of four parts. The first part defines quality and software quality through a review of several definitions provided by researchers and organizations. Quality control and how it may be applied to software are discussed in the second part which also includes an analysis on the feasibility of testing safety-critical software. The third part presents software quality assurance along with its activities and tools. The industry and several organizations have developed standards and maturity models aimed at software quality. In the fourth part of this chapter the two most commonly used maturity models (CMM and SPICE) are reviewed followed by a summary of standards developed for safety-critical software leading a to a description of DO-178B which is the core standard behind this thesis and in use in the aerospace industry.

3.1 Definition of Quality

The first step towards understanding software quality is defining it. In order to do so, a definition of quality in a universal sense is required. Through a review of the work of several quality gurus and experts, the following sections first define quality in general then software quality, and finally, quality as a set of dimensions.

3.1.1 General Definition of Quality

The word quality is widely related to the "degree of excellence" of a product or service. However, defining quality in a simple form or expression is surprisingly difficult and is sometimes based on personal experience, and it is often defined in different ways. Juran defined quality as "fitness for use" (Juran, 1951), while Crosby defined it as "conformance to requirements" (Crosby, 1980). Feigenbaum (1991) called quality "the total composite product and service characteristics of marketing, engineering, manufacturing and maintenance through which the product and service in use will meet the expectations of the customer".

The International Organisation for Standardisation (ISO) has defined quality in its ISO 8402 standard as "the totality of features or characteristics of a product or service that bear on its ability to satisfy stated or implied needs" (ISO, 1994). Moreover, quality is sometimes quantified as the ratio of performance over expectations that are "based on the intended use and the selling price" (Besterfield, 2001):

$$Q=P/E \tag{3.1}$$

where Q = quality

P = performance

E = expectations

This approach makes quality a subjective issue since customer needs and satisfaction vary from one person to another.

Taguchi had a quite different approach in defining quality. According to him, “an article of good quality performs its intended functions without variability, and causes little loss through harmful side effects, including the cost of using it" (Taguchi, 1986).

Now that quality has been defined in its general sense, it would be easier to understand quality in software products. The following sub-section is dedicated to the definition of Software Quality.

3.1.2 Definition of Software Quality

In their book Definitions in Software Quality Management, Fisher et al. (1979) defined software quality as “the composite of all attributes which describe the degree of excellence of the computer system”. Reifer (1985) further developed this definition by stating that software quality is “the degree to which a software product possesses a specified set of attributes necessary to fulfill a stated purpose”.

In the Institute of Electrical and Electronics Engineers (IEEE) Standard Glossary of Software Engineering Terminology (2002), software quality is defined as “the totality of features and characteristics of a software product that bear on its ability to satisfy given needs; for example conform to specifications”. This definition extends to Crosby’s definition of software quality: “conformance to requirements”. In the same glossary, the IEEE includes the customer in a sub-definition of software quality: “The degree to which a customer or user perceives that software meets his or her composite expectations”.

IEEE/EIA 12207 was adopted for use by the Department of Defence in 1998, replacing MIL-STD-498 which was regarded as the main software development standard in the United States and throughout the world; however, the definition of software quality remained as "the ability of software to satisfy its specified requirements" (IEEE/EIA 12207).

Given all the previous work on defining quality and software quality, the best approach to understand product quality –whether hardware or software- remains handling it as a set of dimensions as presented in the following sub-section.

3.1.3 Quality Dimensions

Garvin (1987) suggested that quality may best be defined as a set of dimensions that add value to the product. Garvin introduced the following quality dimensions:

- Performance: Relates to the primary operating characteristics.
- Features: Refers to the secondary characteristics and added features or options.
- Conformance: Extent to which the product meets specifications or standards set by the industry or client.
- Reliability: This concept is concerned with the consistency of product performance over time. Statistically, it can be seen as the average run time before failure.
- Durability: This dimension refers to the useful life of the product; the length of time over which it can be used before replacement.

- Serviceability: This aspect includes the ease of repair, resolution of problems and complaints, and customer service.
- Aesthetics: This aspect is concerned with the sensory characteristics of the product, such as its look or exterior finish.
- Perceived Quality: This dimension is better known as the reputation of the product or its supplier, past performance and other intangibles, such as being ranked first.

Because of software products' special characteristics and own functions, some dimensions must be added to the ones suggested by Garvin. Quality dimensions related to software products were described as follows by Boehm et al. (1980) as:

- Modifiability: This aspect is concerned with the ability to have enhancement and changes made easily.
- Understandability: This dimension refers to the ability to understand the software in order to change or modify it.
- Portability: This is also known as compatibility; the ability to move the software easily from one environment to another.
- Efficiency: Software efficiency includes performance speed, memory management, compactness, and the algorithm logic.
- Usability: This is intended to evaluate the user friendliness of the software.
- Testability: This targets the ability to construct and execute test cases easily.

3.2 Quality Control

Quality Control existed ever since products existed: people always tended to compare products, check for better ones, and verify if a product is good or not. However, "Quality Control" in this thesis refers to the "Science of Quality Control" which was pioneered in the 1920s by Walter Shewart (1931) when he introduced statistics, analysis and measurements as quality control techniques in manufacturing processes.

Modern Quality Control knew its rise during World War II based on statistical methods such as random sampling techniques, testing, measurements, inspection, defective cause findings, and acceptance criteria. During the war, military products' suppliers were mandated to employ statistical quality control in their manufacturing processes. Since then, Quality Control has been widely employed in all manufacturing fields and also became a competitive weapon enabling organizations to survive in modern marketing.

The most common definition of quality control is provided by quality guru J.M. Juran in which "Quality Control is the regulatory process through which we measure actual quality performance, compare it with standards, and act on the difference" (Juran et al., 1979). Deming said: "Quality Control does not mean achieving perfection; it means the efficient production of the quality that the market expects" (Deming, 1986). In the Telecom Glossary 2000 developed by the Alliance for Telecommunications Industry Solutions (ATIS) and approved by ANSI, Quality Control is "a management function whereby control of the quality of raw materials, assemblies, produced material, and components, services related to production, and management, production, and inspection processes is exercised for the purpose of preventing undetected production of defective material or the rendering of faulty services" (ATIS, 2000 and ANSI, 2001).

Just like all products, Quality Control can also be applied to software. Fisher et al. (1979) define Software Quality Control as "the assessment of procedural and product compliance. Independently finding these deficiencies assures compliance of the product with stated requirements." According to Reifer (1985), "Software Quality Control is the set of verification activities which at any point in the software development sequence involves assessing whether the current products being produced are technically consistent and compliant with the specification of the previous phase." Narrowing those two definitions down, Schulmeyer (1999) defines Software Quality Control as the "independent evaluation to assure fitness for use of the total software product".

3.2.1 Statistical Quality Control

Statistical Quality Control consists of using statistics methods to analyse and study a certain production process, or the whole organization, or a product output. These methods take as input measurements tallies collected by inspections and generate statistical information about the data being measured. Computing the mean and standard deviation of the diameter of a set of ball bearings produced is a simple example of the application of statistical methods; the computed mean and standard deviation can be compared to the design requirement to accept or reject the set. Statistical Methods are also used to track manufacturing processes; historical data analyzed statistically give a picture if the quality produced by these processes has improved, weakened, or remained unchanged.

3.2.2 Quality Control Tools

Over the years several tools have been developed to assist quality engineers in improving and managing quality in their organizations. Most of these tools are based on inspections, analysis, and measurements. The following table (3.1) lists of the most commonly used tools.

Table 3.1: Quality Control Tools

Tools	Their use
Run Charts	Used to display process performance with reference to time.
Radar Charts	Multiple axes charts used to plot data when several different factors relate to the same item.
Scatter Plots	Used to study if any relationship exists between two variables by plotting one versus the second.
Histograms	Graphical summary of the distribution of a data set. Individual data points are grouped together in classes, and the frequency of occurrence of each class is displayed.
Pareto Charts:	Used to display the Pareto principle in action. Pareto charts are histograms sorted by descending order of each class' frequency of occurrence.
Control Charts	Used to plot performance data over time with regard to a Lower Control Limit, an Upper Control Limit, and a Center Line.
Cause & Effect Diagram	Created by K. Ishikawa, cause and effect diagrams are used to search for root causes, identify areas where there may be problems, and

	compare the relative importance of different causes.
Flow Charts	used as graphical representations of the process where the steps are shown with symbolic shapes, and the flow is indicated with arrows connecting the symbols

The general methods used in statistical analysis and interpretations of data include the following:

- Quantitative calculations such as Mean, Variance, Standard Deviation, Correlation, Defect Rate, Sample Size, Process Capability Calculations, etc.
- Probability functions and distributions such as Normal, Binomial, Uniform, Poisson, Gamma, etc. Some of their practical use is in simulation studies, to calculate confidence intervals for parameters, and to calculate critical regions for hypothesis tests.
- Fitness Tests used to verify if a sample of data comes from a population with a specific distribution. The Anderson-Darling Test, the Chi-Square Goodness of Fit test, and the Kolmogorov-Smirnov Goodness of Fit test are examples of Fitness Tests.

3.2.3 Applying Quality Control to Software products

As mentioned earlier, all Quality Control methods rely on quantitative inputs obtained from measurements and tests. Measurements are relatively easily made in physical products (hardware) since they mostly consist of measurable metrics such as length, weight, time, temperature, or logical metric (true or false). However, software products cannot be measured using those tangible metrics. This raises two important problems: how to measure software and how to apply Quality Control methods to safety-critical software products?

In order to answer those two questions, the following sections first look into work done on software measurement and identifies metrics that are used in the case of safety-critical software. Then, the feasibility of testing safety-critical software for Quality Control is analyzed.

3.2.4 Software Quality Measurement

Measurement in science has a long tradition; physicist Lord Kelvin once said "when you can measure what you are speaking about, and express it into numbers, you know something about it; but when you cannot measure and express it in numbers, your knowledge is of a meagre and unsatisfactory" (Kelvin, 1891-1894).

The first attempt to create a method for software quality measurement appears to be the work of Rubey and Hartwick (1968) who considered quality factors to be based on elementary code features that are assets to software quality. They defined "attributes and metrics" for software code. An "attribute" is a prose expression describing a particular quality desire. A "metric" is a mathematical function that relates to the attribute. Rubey and Hartwick defined seven high-level attributes, as for example: "Attribute 1: mathematical calculations are correctly performed; Attribute 5: the program is intelligible; and Attribute 6: the program is easy to modify". Each of the seven high-level attributes was refined into six to twelve “sub-attributes”, however only a few metrics were defined. The presence of each attribute in the software is ranked on a scale of 0 to 100 which gives a numerical expression of the conformance of the software with the attributes. According to Rubey and Hartwick this can be used as a metric to measure and control software quality.

Wulf (1973) also recognized the importance of characterizing and dealing with attributes in terms of software quality. He identified seven non-overlapping attributes: robustness, cost, human factors, maintainability/modifiability, portability, clarity, and performance. As the trend in software use became widespread, more people emphasized software quality and the importance of establishing good programming practices.

Fagan’s (1976) work is the most known methodology for software inspection. It consists of seven steps: planning, overview, preparation, inspection, process improvement, re-work, and follow-up activities. The goals are to detect and remove errors in the work products. Fagan maintained that “there is a strong economic case for identifying defects as early as possible, as the cost of correction increases the later the defect is discovered” (Fagan, 1976).

3.2.5 Safety-Critical Software Quality Measurement

The interest in safety-critical software products is to have an output conforming to all design requirements. A failure in producing such an output may lead to tragic consequences on human life. Therefore the target is to have zero failure.

Failure can be measured by a failure rate (*θ*) which is the ratio of the number of defective units in a population to the total number of the population. In software, a defective unit is a defective output. The defect rate is expressed mathematically as follows:

$$\theta = \frac{D}{N} \tag{3.2}$$

where	θ	is the defect rate
	D	is the total number of defects in the population
	N	is the size of the population

In probability, statistics, and quality control studies, the defect rate (*θ*) has always been related with the binomial distribution. The binomial distribution is used when there are exactly two mutually exclusive outcomes of a trial, either a success or a failure. The term "trial" refers to each time a physical item is tested to check if it is defective or not. When the product tested is software, "trial" refers to each time an input is given to the software program/function in order to verify the output. The formula of the binomial distribution is given by:

$$P(x) = b(x) = \binom{n}{x}\theta^{x}(1-\theta)^{n-x} \tag{3.3}$$

Equation 3.3 gives the probability of getting x failures in a sample of n units taken from a population having a defective rate of *θ*.

Given the zero failure requirement for safety-critical software products, a product S fails when x>0. Let S' denote a failed product S, using equation 3.3 the probability of failure *P(S')* of safety-critical software products can be computed as follows:

$$\mathrm{P}(\mathrm{S}') = P(x > 0) = 1 - P(S) = 1 - P(x = 0) = \binom{n}{0}\theta_S{}^0(1-\theta)^{n-0} = 1-(1-\theta_S)^n$$

$$P(S') = 1 - (1 - \theta_S)^n \tag{3.4}$$

where: θ_s is the failure rate of the whole product

n is the number of inputs tested

A software product *(S)* is a set of several subroutines and functions *(s)*. Based on equation 3.4, the probability of failure of a certain subroutine s_i ($P(s_i')$) can be expressed by the following equation:

$$P(s_i') = 1 - (1 - \theta_{s_i})^{n_{s_i}} \tag{3.5}$$

where: θ_{si} is the failure rate of the subroutine

n_{si} is the number of inputs tested

The failure of a safety-critical software product *(S')* occurs when at least one of its composing subroutines or functions fails *(s')*. Therefore, P(S') is true if any P(s_i') is true for *i=1...K.*

A general formula for P(S') in terms of s_i can be derived when s_i's are independent from each other:

$$P(S') = 1 - (1 - \theta_{s_1})^{d_1 n_{s1}} (1 - \theta_{s_2})^{d_2 n_{s2}} \ldots (1 - \theta_{s_k})^{d_k n_{sk}}$$
$$P(S') = 1 - \sum_{i=1}^{k} (1 - \theta_{s_i})^{d_i n_{s_i}} \tag{3.6}$$

The formula can now be used to derive all parameters required for the analysis of the feasibility of testing safety-critical software.

3.2.6 Analysis of Safety-Critical Software Testing

Testing is fundamental in product quality control. Most quality control methods are based on data and statistics collected from testing the products and/or their parts. A question was raised in Section 3.2.3 on how to apply quality control to safety-critical software. In order to answer this question it is important to check whether safety-critical software can be tested or not.

In the following an analysis first looks at how long it will take to test safety-critical software products by means of two approaches: Time Functions and Growth Models. Then, the analysis investigates the assumptions of subroutines independencies and failure replacement and their effect on testing time if any.

3.2.6.1 Testing Time

3.2.6.1.1 Using Time Functions

The customary way of checking the performance of a software product is through testing. The interest is to find defects and report them to be fixed. Therefore, the testing time is the same as the expected run time to failure (R_t). Safety-critical software products "are designed to attain a probability of failure ($P(S')$) on the order of 10^{-7} to 10^{-9} for 1 to 10 hour missions" (Butler and Finelli, 1993). A $P(S')$ within this small range will lead to an extremely long testing time as illustrated in the following:

The Expected Run Time to Failure (R_t) is the estimated length of time the software will run before *f* failure occur in *l* loop or iteration.

$$R_t = \mathrm{M}_t \frac{f}{l} \tag{3.7}$$

where M_t is the mean time to failure (i.e. the average time that elapses before finding the first failure). Given equation 3.8 of *P(S')* in terms of M_t as derived in Appendix A:

$$P(S') = 1 - e^{-\frac{t}{\mathrm{M}_t}} \tag{3.8}$$

then R_t can be derived as a function of *P(S')* where:

$$R_t = \frac{t}{-\ln(1 - P(S'))} \frac{f}{l} \tag{3.9}$$

If *P(S')* is 10^{-9} for a 10 hours mission (*t=10*) then the testing time to find the first failure (*f=1*) in 1 loop (*l=1*) will be almost equal to 1141550 years! It will take 1000000 loops (*l*) to decrease the testing time to about 1.15 years!

The next section checks if same results are obtained when using Growth Models to compute testing times.

3.2.6.1.2 Using Growth Models

"In growth models the cause of failure is determined; the program is repaired and is subject to new inputs. These models enable [the estimation of the] probability of failure of the final program" (Butler and Finelli, 1993). Nagel and Skrivan (1982) experimented with growth models in order to determine an expected testing time for six different software programs. They derived a log-linear model from which estimation can be made for the time needed to reach a certain probability of failure ($P(S')$). The result was even longer than what has been computed in 3.2.6.1.1 when $P(S') = 10^{-9}$. Littlewood writes that "clearly, growth techniques are useless in the face of such ultrahigh reliability requirements. It is easy to see that, even in the unlikely event that the system had achieved such a reliability, we could not assure ourselves of that achievement on an acceptable time" (Littlewood, 1989).

3.2.6.2 Independency

So far, only the case where all subroutines and functions (s_i) composing the end software product (S) being independent from each others has been considered. All formulas used took advantage of the independency property of an intersection of two probabilities ($P(s_1' \cap s_2') = P(s_1') \times P(s_2')$). This property simplified calculations. In software, it is common to have interdependent subroutines and functions. When the independency does not hold, the computations performed still hold since $P(s_i')$ is also required to be extremely small.

3.2.6.3 Failure Replacement

When using the binomial distribution, an assumption is made that each time a failure is found it is fixed (or replaced) and the sampling continues into the next failure. When sampling is done without replacements, the geometric distribution should be used. However, as demonstrated by Butler and Finelli the "expected [testing] time with or without replacement is almost the same" (Butler and Finelli, 1993).

3.2.6.4 Analysis Conclusion

Software does not physically fail as hardware does. Physical failures occur when hardware wears out, breaks, or is adversely affected by environmental phenomena. Software is not subject to these problems. Software faults are present at the beginning and throughout a system's life time; they are the product of human "improper" reasoning. It has been shown that performing tests on safety-critical software products is infeasible given the ultrahigh reliability requirements for such products. In addition, a conclusion can be made that Quality Control techniques in their mathematical and statistical forms cannot be applied to safety-critical software products. Therefore, it is necessary that credible procedures be developed to ensure that quality and reliability are built into safety-critical software products right from the first stage of their lifecycle. These procedures are better known as Quality Assurance.

3.3 Software Quality Assurance

The American National Standards Institute (ANSI) and the Institute of Electrical and Electronics Engineers (IEEE) define Software Quality Assurance (SQA) as "a planned and systematic pattern of all actions necessary to provide adequate confidence that the software product conforms to established technical requirements" (ANSI/IEEE, 1981). This definition stresses that actions taken should be "planned and systematic" in order to achieve quality with "confidence". The philosophy championed is "Prevention of non-conformities from the start rather than Detection at the end" (Kumar, 1994). This prompts the need for a procedure assuring that a product conforms to functional and design performance criteria during all levels of the development process; this procedure is to prevent non-conformities and detect the root cause of possible errors while making sure they will not recur. Accordingly, "software quality assurance lays considerable stress on getting the design right prior to coding" (Manns et al., 1988).

Over the years, the software industry witnessed a high percentage of projects going over budget and/or over schedule. In 1995, U.S. companies spent an estimated $59 billion in software project cost overruns and another $81 billion on cancelled software projects (Johnson, 1995). In another survey of 72 information system development projects in 23 major U.S. corporations, the average effort overrun was 36% and the average schedule overrun was 22% (Genuchten, 1991). Given this fact and that software development teams tend to sacrifice quality in order to meet budget and schedule, Galin (2004) rewrote the SQA definition given by ANSI and IEEE to include budget and schedule concerns. Accordingly, SQA is "a systematic, planned set of actions necessary to provide adequate confidence that the software development process or the maintenance process of a software system product conforms to established functional technical requirements as well as with the managerial requirements of keeping the schedule and operating within the budgetary confines" (Galin, 2004).

SQA activities are usually the responsibility of one person, an SQA team, or a whole department depending on the organization and the project. Typically those activities include (O'Regan, 2002):

- SQA Planning: prepare a software quality assurance plan which interprets quality program requirements and assigns tasks, schedules and organizational responsibilities.
- Software testing surveillance: report software problems, analysis of error causes and assurance of corrective action.
- Software verification: verify the correctness of the software.
- Software validation: validate that software complies with the requirements.
- Software development process: audit the development process.
- Quality promotion: promoting quality and its importance in the organization.
- Certification: prepare the software product to pass any needed certification requirements.
- Records keeping: keep design and software problem reports, test cases, test data, logs verifying quality assurance reviews and other actions.

The SQA team is independent from the software development team; people building the software cannot be the same as those checking its quality.

3.3.1 SQA Tools

Software Quality Assurance is extremely important in the world of safety-critical software. The success of such a product and the success of the organization producing it highly depend on the high quality of the product and hence on the SQA implemented. Therefore, tools are required to monitor the SQA plan and ensure that it is leading to its goals. Moreover, these tools can be used by the organization to compare its SQA processes between different projects, monitor its SQA improvements from a project to another. It is essential for software organizations to learn from mistakes made in order to prevent them in the future, SQA tools can help with that too.

The following provides an overview of different SQA tools that can be applied in safety-critical software production environment. Those tools can be grouped into two categories: Error Management Tools, and Managerial Tools.

3.3.1.1 Error Management Tools

Error Management Tools are tools used to collect and analyze data related to the occurrence and detections of errors, faults, and defects. For safety-critical software products, the most relevant Error Management Tools are: Error Type Pareto Chart, Error per Stage Chart, Number of Repetitions Till Error Free (and its variants), and Time Till acceptance.

In all of the following sections "error" refers to all errors, defects, faults and failures that can occur throughout the lifecycle of the software being produced.

3.3.1.1.1 Error Type Pareto Chart

In the early 1900s, Italian economist Vilfredo Pareto studied the distribution of wealth in his country and gave a mathematical conclusion saying that about 80% of the wealth is owned by about 20% of the people. The Pareto Principle was introduced when in the 1940's, Joseph Juran "noticed that the concept that a vital few members of the assortment account for the most total effect, and contrariwise, the bulk of the members (the trivial many) account for the very little of the total effect was universal" (Juran et al., 1979). Hence, concentrating improvement efforts on these few will have a greater impact and be more cost-effective than undirected efforts.

Since measurements are not feasible in safety-critical software products, the Pareto Principle can be applied by classifying errors in different error types (and sub-types when required).

Categories of error types may differ from one organization to another. Each SQA team can define its own classification according to the project at hand. However, for purposes of comparison and assessment, it is recommended that the organization adopt the same error category definition for most of its projects. The following table (Table 3.2) is an example of error categories that can be used as suggested by Dunn (1987):

Table 3.2: Examples of error categories

Requirements Errors	o Incorrect Reflection of Operational Environment
	o Incomplete Requirement
	o Infeasible Requirement
	o Conflicting Requirements

	o Software Requirements Specification Inconsistent with Other Specifications o Improper Description of the Initial State of the System
Design Errors	o Range Limitations o Infinite Loops o Unauthorized or Incorrect Use of System Resources o Computational Error Improperly Analyzed o Conflicting data Representations o Software Interface Anomalies o Defenceless Design o Inadequate Exception Handling o Non-Conformance to Specified Requirements
Coding Errors	o Misuse of Variables o Mismatched Parameter Lists o Improper Nesting of Loops and Branches o Undefined Variables and Initialization Defects o Missing Code o Incorrect Access of Variable

Dunn's error classification categories are given for general software products. In the case of safety-critical software, an error category should be added for errors occurring because of an inconsistency with the standards required for certification.

When information on all errors detected is collected and all errors classified into their respective category, the Error Type Pareto Chart (ETPC) is built by plotting the cumulative frequencies of the relative frequency data (event count data), in descending order. When this is done, the most essential factors for the analysis are graphically apparent, and in an orderly format.

ETPC is extremely useful to identify those factors that have the greatest cumulative effect on the system, and thus screen out the less significant factors in an analysis. Ideally, this allows focusing attention on a few important factors in a process; as mentioned earlier, concentrating improvement efforts on these few will have a greater impact and be more cost-effective than undirected efforts. Moreover, ETPCs from different projects allow the organization to track the evolution of errors and which factors/processes are generating the highest number of errors.

3.3.1.1.2 Errors per Stage Chart

The purpose of the Errors per Stage Chart is to check during which stage of the software development flow most of the errors are occurring.

The Error per Stage Chart is based on counting for each development stage the number of mistakes that caused errors. Each time an error is found or occurs, it is inspected to find its cause or causes. Causes can be related to the same stage or different ones. Each cause increases by one the number of mistakes for its corresponding stage or stages. When all the counts are completed the chart is plotted.

One variant of the Error per Stage Chart is the Weighted Error per Stage Chart where errors are given a certain weight given their importance and their effect on the end software-product.

The Errors per Stage Chart is used to monitor each stage during software development and check at what stage most sources of errors are. This information is very important for the organization for its quality management improvement policy. In addition, Errors per Stage Charts from different projects can tell the organization if its improvement policy is working or not.

3.3.1.1.3 Number of Repartitions till Error Free

It is important for the organization to know how many times a certain process or development procedure is repeated till the error is fixed; hence the purpose of the Number of Repartitions till Error Free (NREF).

The NREF is a simple tally which is increased by one every time a certain process or a certain development procedure is repeated with the goal of fixing a given error. Even if a procedure was repeated by mistake or unintentionally, the tally is increased. Several variants of NREF exist and are summarised in Table 3.3.

Table 3.3: NREF Variants

Variant	Description
Time till Error Free	The Time till Error Free (TEF) is an NREF reflected by a time unit. The tally is increased by the number of hours (or any other time unit) required during each repetition performed until the given error is fixed.
Total Number of Repetitions till Error Free	The Total Number of Repetitions till Error Free is the summation of all NREF for all procedures repeated in order to fix a given error.
Total Time till Error Free	The Total Time till Error Free is the summation of all TEF for all procedures repeated in order to fix a given error.
Total Number of Repetitions till Error Free per Error Type	The Total Number of Repetitions till Error Free per Error Type is the summation of all Total Number of repetitions till Error Free for all errors that fall in the same error type classification.
Total Time till Error Free per Error Type	The Total Time till Error Free per Error Type is the summation of all Total Number of Repetitions till Error Free for all errors that fall in the same error type classification.
Total Number of Repetitions till Error Free per Procedure	The Total Number of Repetitions till Error Free per Procedure is the total number of times a certain procedure was repeated in order to fix all errors.

Total Time till Error Free per Procedure	The Total Time till Error Free per Procedure is the total time spent repeating a certain procedure in order to fix all errors.
Averages	All of the seven tallies described above can be averaged to the total number of errors or procedures in a software development project.

The Number of Repartitions till Error Free and its Variants can be used to reflect how much time (in time units), resources and effort are lost on fixing errors during a software development project. These calculations –especially when averaged- can also be used to compare projects and monitor improvement policies made by the organization.

3.3.1.2 Managerial Tools

The Managerial Tools are used by the SQA team to manage and control the progress of the safety-critical software development project. In the following, the main three Managerial Tools are discussed: Software Configuration Management, Auditing, and SQA Project Management Tools.

3.3.1.2.1 Software Configuration Management (SCM)

During software development, changes occur. Uncontrolled changes to the software under development "are usually a significant cause of changes to a project's schedule and budget; unmanaged change is the largest single cause of failure to deliver systems on time and within budget" (NASA, 1995). Hence, a management system and set of activities is required to control changes by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling changes that are imposed, and auditing and reporting on the changes that are made.

Bersoff et al. (1980) described SCM as "the discipline of identifying the configuration of a system at discrete points in time for purposes of systematically controlling changes to this configuration and maintaining the integrity and traceability of this configuration throughout the system life cycle". The IEEE defines SCM as "a discipline applying technical and administrative direction and surveillance to identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements" (IEEE, 2002).

The first step in SCM is to identify all items of the software project that might change. These items are called Configuration Items (CI). According to IEEE, CIs are "an aggregation of hardware, software, or both that is designated for configuration management and treated as a single entity in the SCM process. They can be a program, a group of programs, a component library, a function, a subroutine, project documentation, a user manual, test plan, test data, project data, and so on" (IEEE, 2002).

Each time a CI is developed it is sent to the SCM team in order to be checked and saved in the control library. A baseline is a set of CIs in the control library that are related to the same development phase or activity. For example, a design baseline groups all CI's developed during the Design phase. IEEE defines baseline as "a specification or product that has been formally reviewed and agreed on, which thereafter serves as the basis for further development and which can be changed only through formal change control procedures" (IEEE, 2002). Hence, a baseline, together with all approved changes, represents the current approved configuration of the software project.

The essential role of SCM is to control all changes accruing in all CIs. Uncontrolled changes may result in very bad effects on the course of the software project especially on the budget and schedule. Control processes are Version and Variants Control, Change Control, and Release Engineering.

A version is an initial release or re-release of a CI. Versions cannot be identical; each version may differ in performance, characteristics, or functionality. Versions may be the result of fixing a bug in a previous version. When versions have equivalent functionality but are designed to be integrated in different hardware and/or software environments, they are called variants.

Versions and Variants Control is the process that deals with controlling all changes made to all project CI versions or variants. This process also deals with version naming. A version name should be descriptive and designed in such a way as to determine its relative position in the version hierarchy. Change Control is a process implemented by the organization and used to approve changes or reject them. This process defines the procedures a change should follow and who can authorize the change or reject it.

A release consists of more than just executable code; it includes installation files, data files, setup programs, all required documentation, certification documents, new release features document, etc. Release Engineering refers to a controlled way of acquiring approval for release deliverables and establishing the production baseline so that the end-product can be made available to customers.

3.3.1.2.2 Auditing

Auditing processes are the tools by which an organization can ensure that software development is being performed in conformance with standards, guidelines, regulations, and requirements.

Usually, an audit is performed at the end of each development phase and before proceeding to the following phase. In the case of safety-critical software projects, the assessment of the Acceptance Criteria at the end of each phase should also be audited.

According to IEEE, audits are "activities performed to independently evaluate the software [...]" (IEEE, 2002). Therefore, audits should not be performed by the same personnel who worked on the development of the phase being audited or who passed the phase for Acceptance Criteria. Audits should be performed with a high degree of objectivity, especially in safety-critical software in order to ensure a high quality product. Within the organization, the Software Quality Assurance team is the most qualified to perform audits. Audits may also be conducted by an external party such as the client or agencies specialized in auditing.

During an audit, discrepancies or anomalies are recorded by the auditors. They are reviewed by the auditing teams and corrective actions are recommended to the developing team. Follow-up actions are required to make sure that all recommended actions have been performed. Audit findings have to be documented properly. In general, a successful audit requires the following:

- An audit plan
- An audit agenda
- Full access to the product being audited and all of its details
- Full access to all documents and applicable specifications (manuals, drawings, etc.)

3.3.1.2.3 SQA Project Management Tools

As an essential component of the software project, SQA activities should be managed properly; a project cannot be successful unless all of its components are controlled and managed effectively.

SQA can rely on project management techniques and tools to bring its activities into control. In general, good project management depends on three factors: time, cost and performance. "Projects are successful if they are completed on time, within budget, and to requirements" (Gardiner, 2005). Hence, tools are required to monitor those three factors in SQA.

3.3.1.2.3.1 Planning

Planning is crucial for SQA. Plans are the simulation of how SQA will satisfy the three factors of time, cost and performance; they aid coordination and communication, provide a basis for control, help avoid problems and draw a path towards meeting all requirements. Plans define each person's (or group of persons') responsibilities and roles within the SQA team and in the whole project. They also define how SQA is to perform its duties and activities; this helps coordination and communication within the organization.

Projects are dynamic as many changes occur during development. Plans should be available defining how to control those changes and how to modify existing plans. All SQA activities have their own requirements and goals. Plans should be clear on how SQA is intending to meet those requirements. The following lists some example of plans to be prepared by SQA at the beginning of a software project:

- General SQA Plan
- Qualification Plan
- Testing Plan
- Acceptance Criteria Assessment Plan (for safety-critical software)
- Auditing Plan
- Verification Plan
- Preparation for Certification Plan (when certification is required)

3.3.1.2.3.2 Progress Monitoring

Once the plans are completed and the project is in development, SQA should monitor the progress, how its plans are being implemented and if costs are within set boundaries. Several tools and techniques are available for this purpose. Gantt Charts are useful to verify if the progress is running to schedule and if all scheduled activities have been performed (or are being performed). Earliness or tardiness may signal a fault and schedule updates are required.

Activities flowcharts make it easier for SQA personnel to know what should be done at the current time and to anticipate coming activities, especially those requiring setups and preparation.

All SQA minutes, activities, and progress are to be reported neatly. Reports assist the SQA team to assess its progress and monitor its financial status. Reports help investigating faults found in order to know if they are the result of a planning mistake or a development one. Faults can be an earliness, a tardiness, or an over/below budget activity. Learning from reports is essential for SQA's experience and maturity from a project to another.

3.4 Standards and Models

Since the early 1970s, several institutes, associations, and governmental organizations tackled the importance of software quality by developing standards and quality assurance methods. Those organizations include: the US Department of Defence (DOD), the US

Federal Aviation Administration (FAA), the Institute of Electronic and Electrical Engineering (IEEE), and the North Atlantic Treaty Organization (NATO). The first major standard, entitled "Software Quality Program Requirements", was released by the US Army in 1975 and; it is better known by its reference name: MIL-S-52779. Every standard and guideline developed since then was influenced by this landmark standard. In general, the two most common and widely used standards and models are the Capability Maturity Model (CMM) and SPICE, which are presented in the following followed by a review of standards developed specially for safety-critical software.

3.4.1 The Capability Maturity Model

The Capability Maturity Model (CMM) is a process maturity model which enables software development organizations to define and evolve their processes. The main motivation behind the development of CMM was the need of the Department of Defence (DOD) to develop a mechanism to evaluate the capability of the software contractors. CMM was mainly influenced by the Crosby's quality management maturity grip which describes five evolutionary stages in adopting quality practices. The Software Engineering Institute (SEI) released the first version of CMM (CMM v1.0) in 1991 as a pilot which was revised and released as CMM v1.01 in 1993.

The benefit of CMM is that it allows organizations to follow a logical path toward software development improvement. The CMM evolutionary path is made of five consecutive maturity levels which cannot be skipped. Each level provides a foundation for further improvements. "There is an increase in capability associated with greater maturity, and this enhanced capability is reflected in quality, timeliness of projects, reliability, etc" (SEI, 2000).

The five maturity levels of the CMM path are the initial level, the repeatable level, the defined level, the managed level and the optimizing level. Figure 3.1 illustrates the path of the five maturity levels.

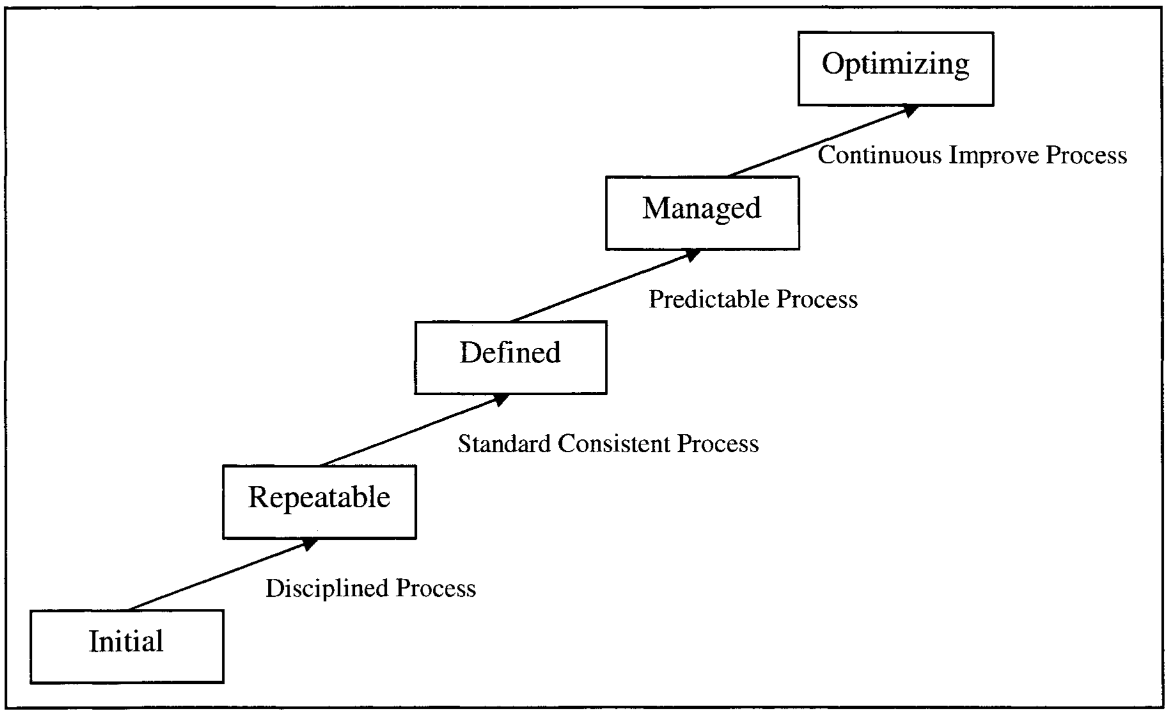


Figure 3.1: CMM Maturity Levels (SEI, 2000)

CMM identifies a cumulative set of 'key process areas' (KPA) which all need to be performed as the maturity level increases. The following provides a brief description of each of the five maturity levels and its corresponding KPA.

- **Level 1 - Initial**

The software process is characterised as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics.

KPA: None

- **Level 2 - Repeatable**

  Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

  KPA: Requirements Management, Software Project Planning, Software Project Tracking and Oversight, Software Subcontract Management, Software Quality Assurance, and Software Configuration Management.

- **Level 3 - Defined**

  The software process for both management and engineering activities is documented, standardised, and integrated into a standard software process for the organisation. All projects use an approved, tailored version of the organisation's standard software process for developing and maintaining software.

  KPA: Organization Process Focus, Organization Process Definition, Training Program Integrated Software Management, Software Product Engineering, Intergroup Coordination, and Peer Reviews.

- **Level 4 - Managed**

  Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.

  KPA: Quantitative Process Management, and Software Quality Management.

- **Level 5 - Optimizing**

  Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

KPA: Defect Prevention, Technology Change Management, and Process Change Management.

Successful implementation of CMM depends on the commitment of the organization; "it requires vision, planning, adequate resources, selection of teams and team leaders, training for teams, and formulation of plans for teams" (SEI, 2000). The implementation is tracked by internal CMM assessments team who coordinates with an external organization specialized in CMM assessments.

SEI has published two new representations of the CMM model known as the "continuous CMMI" and the "staged CMMI" models (O'Regan, 2002). The purpose of those CMMI models is to make CMM compatible with the SPICE standard.

3.4.2 The SPICE (15504) Model

The growth in the number of assessment approaches available was the key motivation behind a study on the need for a software process assessment standard. The study was conducted by a joint technical committee (known as JTC/SC7) between the International Organization for Standardization (ISO) and the International Electrotechnical Committee (IEC). The results of this study drove the establishment of the Software Process Improvement and Capability dEtermination (SPICE) project to standardize and improve on the existing software assessment methodologies. The initial work started in 1990, version 1 was released in 1995, version 2 in 1996, and the first comprehensive version was published in 1998. The current version of SPICE was published in 2005; it is the emerging international standard for software process assessment.

In software production, defects are usually due to defects with a particular development process used to build the software, or due to a faulty execution of a process. SPICE provides a framework for the assessment of software processes and is designed to play a key role in process capability determination and improvement. It allows organizations to understand their key processes and capabilities, as well as to prioritize further improvements consistent with their business goals.

In SPICE, there are 6 levels of capability according to which the software development process are rated. The 6 levels are briefly described in the following:

- **Level 0 - Incomplete Process**

At this level, there is general failure to attain the purpose of the process. There are no easily identifiable work products or outputs of the process.

- **Level 1 - Performed Process**

At this level, the purpose of the process is generally achieved. The achievement may not be rigorously planned and tracked. Individuals within the organization recognize that an action should be performed, and there is general agreement that this action is performed as and when required.

- **Level 2 - Managed Process**

At level 2, the process delivers products of acceptable quality, in conformance to specified standards and requirements, and within defined timescales. Performance according to specified procedures is planned and tracked. The primary distinction from the Level 1 is that the performance of the process is planned and managed and progressing towards a defined process.

- **Level 3 - Established Process**

At Level 3, the process is performed and managed using a defined process based upon good software engineering principles. The process is documented and uses approved and customized versions of standard. The main distinction from Level 2 is that the process is planned and managed using a standard process.

- **Level 4 - Predictable Process**

A process of Level 4 is performed consistently in practice within defined control limits, to achieve its goals. Detailed measures of performance are collected and analyzed. This leads to a quantitative understanding of process capability and an improved ability to predict performance. The primary distinction from Level 3 is that the defined process is quantitatively understood and controlled.

- **Level 5 - Optimizing Process**

At level 5, the performance of the process is optimized to meet current and future needs. The process achieves repeatability in meeting its requirements. Quantitative process effectiveness and efficiency goals for performance are established and continuous process monitoring against these goals is enabled by obtaining quantitative feedback. Improvement is achieved by analysis of the results. The primary distinction from the Level 4 is that the defined process and the standard process undergo continuous refinement and improvement, based on a quantitative understanding of the impact of changes to these processes.

The focus in SPICE is on fixing faulty development processes and improving their application. SPICE is useful for internal process improvement and one of its advantages is that it allows organizations to focus on improvement to select processes related to their business goals rather than step-wise evolution approach of CMM.

CMM and SPICE are currently the two widely used standards and guideline models for software process assessment and improvement. Their successful implementation assists organizations in increasing the maturity and capability of their software development processes.

CMM and SPICE were developed for all types of software production organizations and their integrity has been proven. However, they are too general and wide-ranging, they tolerate small share of defects and errors even at the most advanced levels. This nature of CMM and SPICE makes them weak and ineffective for implementation on safety-critical software production due to the characteristics and extremely high quality and reliability requirements of such products. Therefore, several standards dedicated specially to safety-critical software production have been published. Those standards are reviewed in the following.

3.4.3 Standards for Safety-Critical Software

A number of differing standards for safety-critical software already exist or are currently under development. Most of those standards are designed specifically to a particular industrial sector or even to a type of application within a sector. Mainly, three sectors dominate most of the standards' studies and developments: aircraft industry (particularly airborne software systems), nuclear energy (particularly reactors control software), and railway transport (particularly signalization systems). As an example, the following lists the most relevant standards and their use:

- MALPAS used for the French nuclear plants protection system.
- LDRA Testbed and VALIDATOR used for the assessment of the Teleperm XS, the most widely used software in nuclear power plants.
- IEC 1508, '*Functional Safety: Safety Related Systems*' developed by the International Electrotechnical Commission (IEC) for electrical, electronic, and/or programmable electronic systems.
- The European Norm EN50128 published in 2001 for software in railway applications such as communications, signalling and processing, railway control and protection.
- Defence Standard 00-55 a detailed software standard for safety critical defence equipment developed for the UK Ministry of Defence.
- DO-178B the basic standard for airborne software. It is further detailed in Section 3.4.4

In general, all standards designed for safety-critical software share the following two main tasks:

1. Establishing regulatory requirements for systems and software; i.e., requirements (criteria, norms, rules) important for safety;
2. Assessing the system and software. The purpose of this assessment is to be convinced that the system and software answer established regulatory requirements.

In the field of aircraft related and airborne software development, DO-178B and its updated revisions remain the landmark standard used. DO-178B is reviewed in the following.

3.4.4 DO-178B

The rapid increase in the use of software in airborne systems and equipment used on aircraft and engines in the early 1980s resulted in a need for industry-accepted guidance for satisfying airworthiness requirements. In order to satisfy this need, the Radio Technical Commission for Aeronautics, Inc (RTCA) published in 1982 the document called "Software Considerations in Airborne Systems and Equipment Certification" which is better known by its reference name: DO-178.

In 1983, the RTCA determined that DO-178 should be revised in order to reflect the experience gained in the certification of the aircraft and engines containing software based systems. As a result, a revised version of DO-178 was published in 1985 under the reference name of DO-178A.

However, rapid advances in software technology which were not envisioned by DO-178A and differing interpretations of the guidelines triggered in 1989 the FAA to formally request that RTCA establish a committee for the review, revision, and update of DO-178B. Besides RTCA experts, the committee included personnel from the Airline Pilots Association, the National Business Aircraft Association, the Air Transport Association, ARINC, and the FAA. In 1992, the committee published its final and comprehensive updated version of the standard under the reference name of DO-178B; it became the de facto standard used in airborne software development.

DO-178B establishes software considerations for developers, installers, and users, when aircraft equipment design is implemented using microcomputer techniques.

According to RTCA, the purpose of DO-178B is to provide guidelines for the development of airborne systems' software that "performs its intended function with a level of confidence in safety that complies with airworthiness requirements".

DO-178B is primarily concerned with development processes. As a result, certification in DO-178B requires delivery of multiple supporting documents and records. The quantity of items needed for DO-178B certification, and the amount of information that they must contain, is determined by the level of certification being sought. The targeted DO-178B certification level is either A, B, C, D, or E. Correspondingly, these levels describe the consequences of a potential failure of the software: catastrophic, hazardous-severe, major, minor, or no-effect. The level to which a particular system must be certified is selected by a process of failure analysis and input from the device manufacturers and the certifying authority (Transport Canada, FAA, or JAA), with the final decision made by the certifying authority. Certification at any level automatically covers the lower-level requirement; but, obviously, the opposite is not true. Software certified at Level A can be used in any avionics application.

The DO-178B specification does not contain anything magical; it enforces good software development practices and system design processes. It describes traceable processes for objectives such as:

- High-level requirements are developed
- Low-level requirements comply with high-level requirements
- Source code complies with low-level requirements
- Source code is traceable to low-level requirements
- Test coverage of high-level and low-level requirements is achieved.

DO-178B is treated as a standard that imposes requirements on the development and verification of airborne avionics systems. It is listed by several regulatory bodies as a means of compliance that is acceptable in the aerospace industry and the avionics community.

The European version of DO-178B is called ED-12B; it is published by EUROCAE. This is the same document as the DO-178B and was produced by an international consensus-based committee representing practitioners as well as regulators.

Since the first publication of DO-178B in 1992, several documents such as DO-248B were released in order to clarify the intent of parts of the documents that were misinterpreted or found to be inapplicable. In addition, several updates were released due to new technological improvements and inventions the latest being FAA notice N8110.49 published in 2003.

3.4.5 Criticism of Standards for Safety-Critical Software

There is widespread criticism of current safety-critical software standards. In particular, Fenton and Neil (1998) claimed "software standards are written in such a way that we could never determine whether they were effective or not". They noted the following weaknesses of the standards: “Overemphasis on process rather than product, imprecise requirements, non-consensus recommendations, lack of clarity on risk and benefits, and Standards are too big and poorly organized”. Emmet and Bloomfield (1997) criticized the typical standards formulation process saying that it “was too long and resulted in out-of-date standards [...] and which are often incomplete and ambiguous”. McDermid and Pumfrey (2001) argued that most safety-critical software standards “are not standards compared with standards in other industries and engineering disciplines”.

Those criticisms drive organizations developing standards on a continuous track of revisions and updates in order to improve existing standards or develop new ones. Several standards are currently being studied or are pending approval.

Whilst DO-178B and its clarification notices detail all actions and requirements necessary throughout all stages of the development lifecycle in order to assure the integrity of the software, they do not offer any assessment method to verify that these requirements and actions were met and performed at their corresponding stage before the transition to the following stage in the lifecycle. Hence the objective of the software quality assurance model proposed in this thesis.

3.5 Chapter Summary

In this chapter, software quality was defined as a set of dimensions that add value to the product. The quality level of a software product depends on how it complies with those dimensions.

In order to improve quality, it should be controlled. Quality control techniques, methods, and tools were also discussed in this chapter. Control is based on measurements and testing. The best approach to measure safety-critical software is by quantifying their defective rates. An analysis based on statistics proved that testing such software is infeasible since it will require an extremely long time. Therefore, quality should be built into safety-critical software right from the first stages of development. This can be done by means of a software quality assurance model. Software quality assurance was explained in this chapter with a review of its activities and tools. Software quality assurance models are applied throughout the development lifecycle of software.

The industry and several organizations have developed maturity models and standards in order to help developer to produce high quality software products. In the last part of this chapter two maturity models CMM and SPICE were presented. CMM and SPICE are the most widely used maturity models; they however tolerate a small share of errors in the end-product. This tolerance is relatively high when applied to safety-critical software, which makes the implementation of CMM and SPICE in this field fragile. Hence, several standards have been developed for safety-critical software production. A summary of the most relevant standards was provided in the last part of this chapter followed by a description of DO-178B which is the standard used for airborne software.

DO-178B details all activities and requirements to be performed throughout the development but it doesn't offer any method on how to ensure that they are performed or correctly followed. The SQA model proposed in this thesis offers a method to overcome this problem in SQA.

As mentioned earlier SQA models are applied throughout the development lifecycle. In the next chapter, lifecycles are explained and a lifecycle model for safety-critical software projects is proposed.

4.0 DEVELOPMENT PROCESS

The diverse activities undertaken during the development of software are modelled as a software development lifecycle which begins with the identification of a requirement for software and ends with the formal verification of the developed software against that requirement.

This chapter proposes a development lifecycle modeled by the author specifically for aircraft and avionics safety-critical software projects. The chapter starts with an introduction to development lifecycles explaining why a particular lifecycle is required for safety-critical software projects. After describing the proposed model, the chapter also introduces a proposed method for system verification.

4.1 Introduction to Development Lifecycles

Several lifecycle models for software have been developed and can be classified into three categories: Sequential, Iterative, and Progressive.

Sequential flow of well defined development phases is the traditional approach used to model software development lifecycles. The two classical sequential lifecycles are the "V lifecycle model" and the "waterfall lifecycle model" where phases are represented by a V or waterfall diagram.

Those two models are typically used in projects where the requirements can be identified at the beginning. As shown in Figures 4.1 and 4.2, the V and the waterfall models start with requirements analysis, followed by architectural and detailed design, and then the code is built and tested, which lead to the software and system integration phases. The last phase is performing acceptance tests in order to verify whether the product can be released or not.

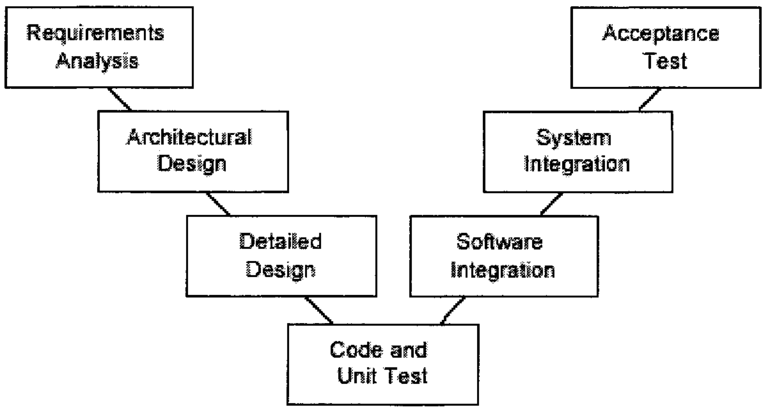


Figure 4.1: V lifecycle model (IPL, 1997)

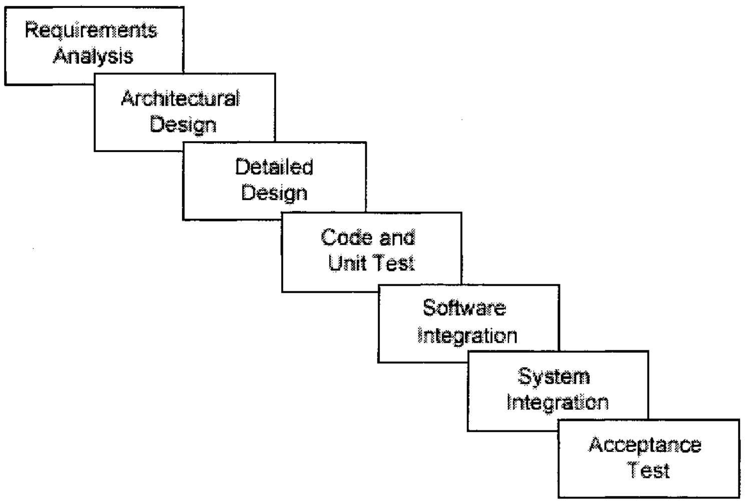


Figure 4.2: Waterfall lifecycle model (IPL, 1997)

Iterative lifecycle models do not attempt to start with a comprehensive specification of requirements. The evolution of the requirements is part of the development lifecycle which is initiated by specifying and implementing just part of the software which can then be reviewed in order to identify further requirements. This process is then repeated, producing a new version of the software after each iteration of the model.

The most common iterative lifecycle is the Spiral Lifecycle Model developed by Boehm. As illustrated in Figure 4.3, the development proceeds in a number of spirals where each spiral typically involves updates to the requirements, design, code, testing, and a user review of the particular iteration or spiral.

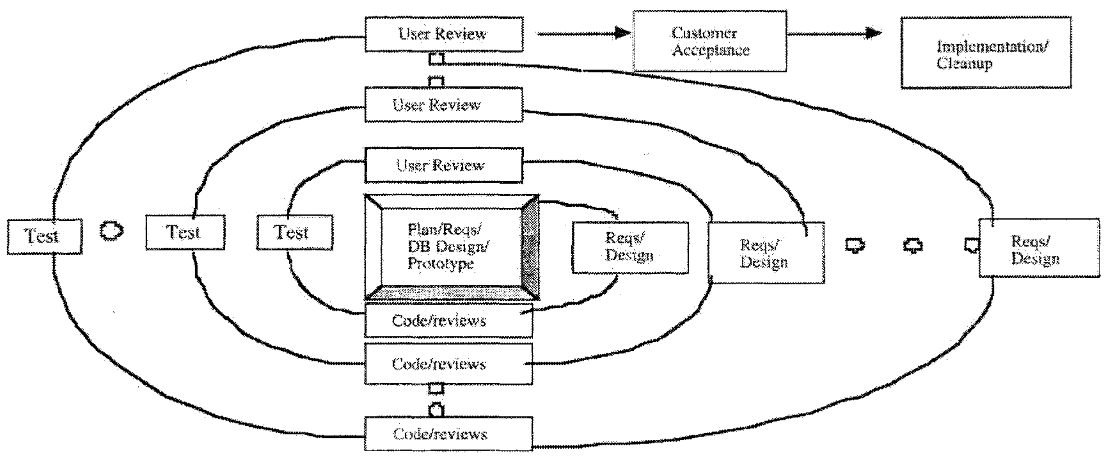


Figure 4.3: Spiral lifecycle model (Boehm, 1988)

Progressive lifecycles are useful when software is needed quickly; they are a compromise between time and functionality. Progressive lifecycles provide "interim deliveries" of the software, with reduced functionality, but serving as stepping stones towards the fully functional software. As illustrated in Figure 4.4, all the phases in a progressive lifecycle are software development that allows the release of a temporary version until the final one is produced. Each phase uses the work of its preceding phase as a start up, or prototype, and improves it by adding functionalities.

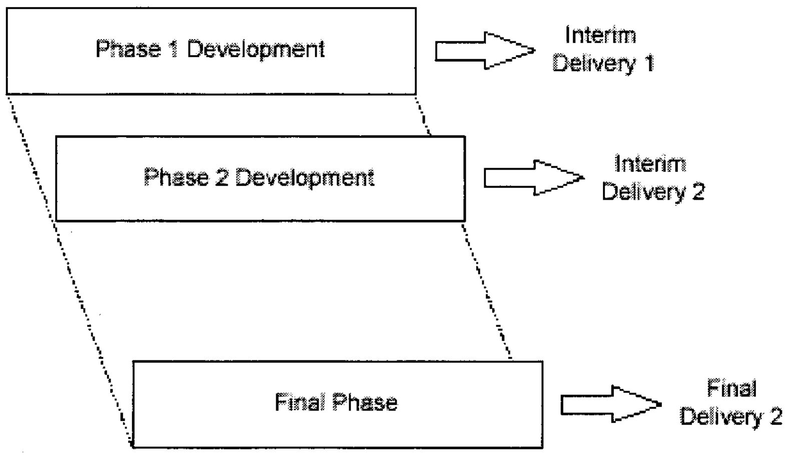


Figure 4.4: Progressive lifecycle model (IPL, 1997)

In the case of safety-critical software, the use of a progressive type lifecycle is not reasonable given the extremely high quality and reliability requirements set for such software. The use of a limited or reduced functionality version may increase the risk of failures. Moreover, the testing and integration of each "interim delivery" will require time and effort. The complexity and relatively large size of safety-critical software projects will require a very large number of development phases and "interim deliveries" that will actually become counter productive and will result in an increased cost and timescale.

Similar to progressive lifecycles, the need for testing at the end of each iteration in iterative lifecycles makes them impractical to use for safety-critical software projects. In addition, in safety-critical software development a change or alteration of a requirement as well as the introduction of new requirements when the software development is already in progress will result in major modifications to all completed work. This translates into undesirable loss of time and resources.

Sequential lifecycles are the most suitable to the characteristics of safety-critical software projects. However, some modifications and adaptations are required to ensure that the cycle flows from a development phase to the next without any risk of having to repeat finished work.

It is also important to note that all lifecycles of all three types (sequential, iterative, and progressive) require at a certain point software testing. However, as it has been shown in Chapter 3, testing is unfeasible for safety-critical software.

The following proposes a development lifecycle modeled typically to safety-critical software development activities. Software testing is replaced by System Verification. A verification procedure for safety-critical software is also proposed in Section 4.3.

4.2 The Proposed Safety-Critical Software Development Lifecycle

The lifecycle proposed in the following is based on the experience acquired during the work on the CRIAQ project and on the software development activities described by DO-178B.

Aircraft and avionics safety-critical software projects go through different stages that can be grouped into three categories: pre-development, development, and post-development. As illustrated in Figure 4.5, the proposed lifecycle models ten safety-critical software development stages in a flow under those three categories. Those ten stages were either taken directly from DO-178B or inspired from its description of a safety-critical software development process. Therefore, in the description of the stages of the proposed lifecycle, DO-178B is referenced whenever a development stage is taken from it.

During Pre-Development, the software project is launched and prepared for development through defining requirements and planning. Development involves all activities taken to build the software product in conformance to all predefined requirements. In Post-Development stages, the software is prepared to be released for use through integration, validation, and certification. The different stages of each category are explained in the following sections.

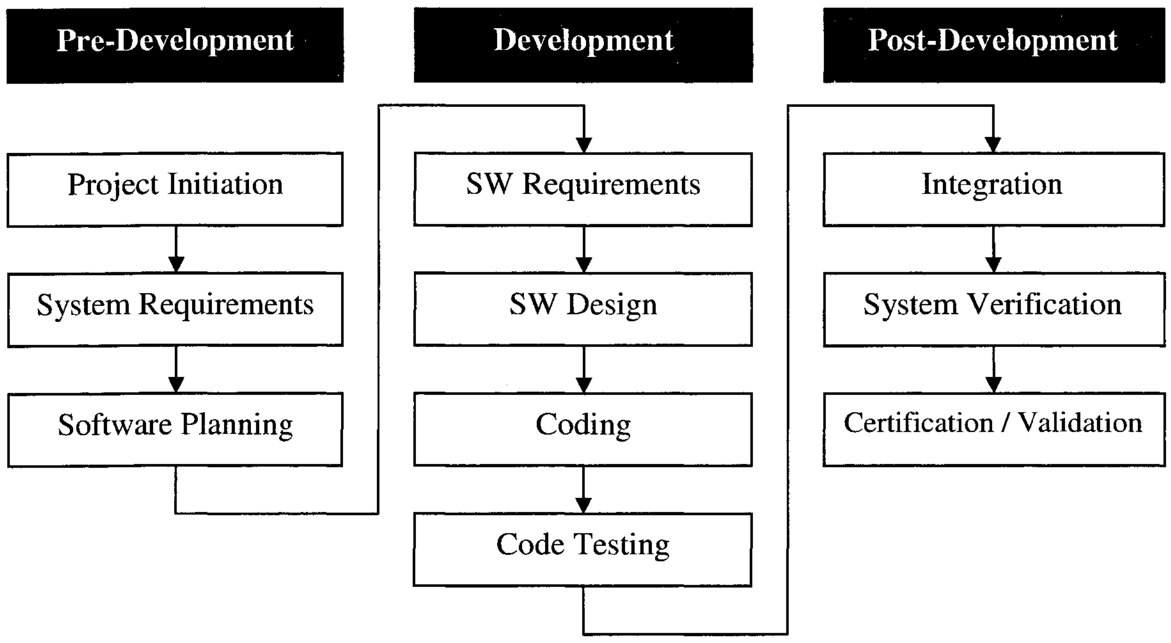


Figure 4.5: Proposed lifecycle model for safety-critical software

4.2.1 Pre-Development

Pre-Development is the first category of stages in the lifecycle of a software project where the latter is launched and initiated. The objectives, potential use, and goals of the software project in question are defined. In preparation for development, requirements for the system within which the software will operate are defined. In addition, "software plans" are prepared describing how and the means by which all project objectives and goals will be met.

4.2.1.1 Software Project Initiation

Software development projects are initiated due to a certain need for new software or a new version of existing software. When this need is materialized, the software project is born. In general, two sides are involved: the client and the developer. The final software is to be delivered to the client. The client can be a person, a company, a department, or a team who requires the development of software in order to complete a certain project. The developer undertakes the software project. Here again the developer can be a person, a company, a department, or a team of people working to ensure the success of the project.

In the case of safety-critical software, a third party is involved in the project: state regulatory bodies (authorities), whose main task is the licensing of activities connected with safety. An assessment of software during certification and licensing is usually carried out on behalf of regulatory bodies either by independent experts, or by experts of organizations which provide a scientific and technical support of regulatory bodies. Software projects are initiated when the client and the developer sit together to discuss the intended project and reach a contract agreement.

During Software Initiation, the purpose and objectives of the proposed project are clearly stated and agreed upon between all parties involved. The functionality, use, and future applications of the product are defined. Usually, those are included in the development agreement contract.

4.2.1.2 System Requirements

Software products are designed to operate or be used on certain systems which can range from personal computers to highly specialized computers such as aircraft embedded systems. Therefore, before starting software design, systems requirements should be defined. System requirements include functionality, objectives, and goals of the system under which the software will be used. Communication between the software and the system is an extremely important element when safety is crucial; communication considerations include: detection of error and corrupted data when loading external information or libraries, compatibility between hardware and software, and compatibility with other software used by the system.

The client and the developer are to define and agree upon the system requirements for the projected software. These requirements should be well defined and stated in the contract. These requirements will serve as the objectives for the developer. Each system-requirement will be used to derive one or more high-level software-requirements during the "Software Requirements" Stage (Section 4.2.2.1). In the case of safety-critical software, the certification and quality level aimed for are an essential part of the system requirements. System requirements should be documented and the document approved by the client and the developer.

4.2.1.3 Software Planning

The third stage in software development is software planning. Its purpose is to define the means of producing software which will satisfy the system requirements. These means are a set of software plans that are defined by DO-178B as follows:

- Software Development Plan

It is a definition and a presentation of the software project lifecycle and the software development environment.

- Software Verification Plan

It is a description of the verification procedures that will be used to check whether the software satisfies the verification objectives.

- Software Configuration Management Plan

It establishes the methods that will be used throughout the project to achieve the objectives of the software configuration management (SCM) process.

- Software Quality Assurance Plan

It defines the methods that will be used to achieve the objectives of the SQA process. The SQA Plan may include descriptions of process improvement, metrics, and management methods.

- Plan for Software Aspects of Certification

This plan is necessary when software certification is required (which is the case for most safety-critical software). "It serves as the primary means for communicating the proposed development methods to the certification authority for agreement" (DO-178B).

This stage also includes the definition of software development environment and test environment, the choice of coding language, compiler, and development standards.

4.2.2 Development

Development groups all stages of the software project lifecycle that together transform the intended software from an idea to an actual product ready to be validated, certified, and integrated into its system. First, Software Requirements are defined based on which developers can work on the Software Design. Once the design is completed, it is converted at the Coding stage from description texts or flow/algorithm charts into code language; then, the code is tested.

4.2.2.1 Software Requirements

At this stage, the software requirements are ready to be derived from the previously defined system requirements. Only High-Level software requirements are developed after analyzing rigorously each system requirement. These High-Level requirements include functional, performance, interface, and safety-related requirements.

Requirements derived at this stage should all be well defined in a Software Requirements Document. In order to ensure traceability, each requirement should be related to the system requirement from which it has been derived. All requirements that can be quantified should be stated with their quantitative terms and tolerances.

Software requirements are the responsibility of the developer. However, the client should agree on them and on the Software Requirements Document before the developer goes into the next stage.

4.2.2.2 Software Design

The objective of the Software Design stage is to design the architecture of the software. During this stage, the previously defined High-Level software requirements are refined through one or more iteration in order to generate Low-Level requirements. Just like the High-Level requirements, Low-Level requirements should be documented and ensured for traceability. The software architecture should be documented in a "Design Description" document including coding algorithms, data flow, and control flow that should all be documented and explained. Design activities could "introduce possible modes of failure into the software or prelude others" (DO-178B).

The final software design should be reviewed to ensure that all requirements have been met. All corrections and modifications made after reviewing the design should be documented and justified.

4.2.2.3 Coding

The purpose of the coding process is to translate the software design from high-level programming language into computer coding language. The language and compiler used should be the same as the previously stated in the Software Planning stage.

The developed code "should be traceable, verifiable, consistent, and correctly implement low-level requirements" (DO-178B).

4.2.2.4 Code Testing

At this stage, the projected software is not ready to be tested and verified yet. However the preliminary tests can be performed on the coded design in order to ensure that desired responses are achieved. Functionalities that can be tested prior to the software integration are also tested.

Test results and corrections required should be documented. Whenever a correction is required in the design, the developer has to go back to the Software Design stage, make necessary adjustments, then correct the code and re-test it.

When all coding tests are satisfied, the software is assumed ready to be integrated into the system.

4.2.3 Post-Development

Once the software is built and found to comply with the design and requirements, it enters the final three stages grouped under Post-Development of the project lifecycle. The first activity is to integrate the product into its system; once this activity is completed, the software is ready to be tested and verified. The final stage is to certify and validate the product by regulatory bodies.

4.2.3.1 Integration

The integration process consists of integrating the designed software as a source code into the system in which it is supposed to function. Integration can be of two types: software integration when software is integrated into another software, and/or hardware integration where the software is integrated into hardware.

Safety-critical software, especially for airborne systems, may include several operational features that are not all used during every application. This can require functions -and hence part of the code- to be deactivated when not needed, mostly for security reasons (avoiding interference). During Integration, it should be proofed that functions are accurately disabled and deactivated for system applications where their use should be avoided.

An Integration Record is required to document the integration process and errors found; it serves as feedback for clarification and correction. Integration is complete when the software is found to be fully operational within its system in accordance with all System Requirements.

4.2.3.2 System Verification

When the software is fully integrated into the system, it is ready to be verified for all requirements and functionalities. At this stage, verification activities are performed on the system as a whole to check the response of the developed software with its system.

In case of errors, necessary corrections can be made directly to the software code or through corrective patches. When a patch is used, "regression analysis [and verification should provide] evidence that the patch satisfies all objectives of the software developed" (DO-178B) and that all errors have been corrected.

System verification is further detailed in Section 4.3 of this chapter along a proposed method which offers a procedure for efficient and thorough verification.

4.2.3.3 Certification and Validation

When the software is found to be performing accurately and meeting all requirements, the developer and the client should work together to prepare the developed software to be certified and/or validated by corresponding authorities.

Lewis (1992) defines validation as "the systematic process of analyzing, evaluating, [and verifying] system and software documentation and code to ensure the highest possible quality, reliability and satisfaction of system needs and objectives". According to the Telecom Glossary 2000, validation is determining "whether an implemented system fulfills its requirements [through] checking data for correctness or for compliance with applicable standards, rules, and conventions" (ATIS, 2000 and ANSI, 2001).

In the same glossary, certification is defined as "the comprehensive evaluation of the technical and non-technical security features and other safeguards of a [software], made as a part of and in support of the accreditation process, to establish the extent to which a particular design and implementation meets a set of specified security requirements" (ATIS, 2000 and ANSI, 2001).

In order to ensure unbiased assessment, Validation and Certification are performed by an organization that is "both technically and managerially separate from the organization developing the product" (IEEE, 2002).

The project lifecycle is from the point of view of the software developing organization. At this stage of the lifecycle, the actual certification is not carried out, it is a process performed by an independent organization once the software is complete. This stage is a preparation for certification; the developer and the client prepare all documents and necessary forms required by the organization responsible of the Validation and/or Certification of the developed software.

4.2.3.4 The Effect of Changes During Development

Changes in requirements might happen during software development and at any stage; in addition, errors occurring and inherited from stage to stage are unfortunately detected late. These are common in software projects and push the development to go back in the flow to an earlier stage in order to fix detected errors or make necessary changes. This creates an iterative cycle. This cycle is known as the Plan-Build-Release cycle. A software project has only one project lifecycle but might have several Plan-Build-Release cycles within its lifecycle. In other words, the Plan-Build-Release cycle is an iterative loop found within the project lifecycle due to error, modifications, and changes in requirements.

Large-scale software projects can be divided into components where each component is developed apart and integrated at the end in the system. This approach requires setting general requirements and design for the whole software at the beginning.

The ad-hoc approach to software engineering is to keep on iterating the Plan-Build-Release sequence until acceptance requirements are met as shown in Figure 4.6.

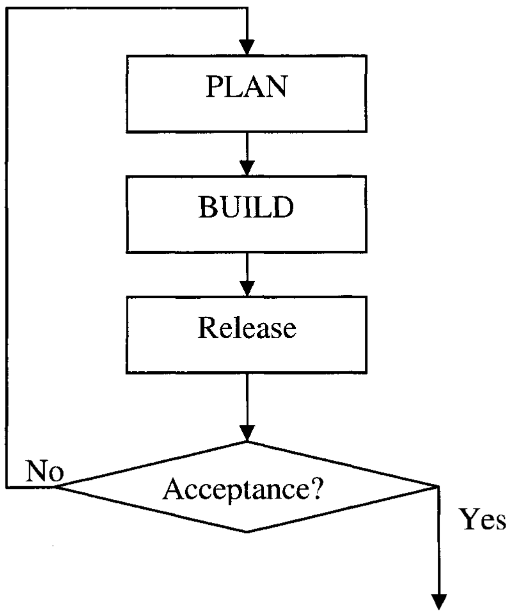


Figure 4.6: Plan-Build-Release sequence

4.2.3.5 Summary

In the previous sections, a lifecycle for safety-critical software projects was proposed, and modeled as a flow of stages through which the software goes from being an idea to a validated/certified ready to use product. Those stages were grouped into three categories: Pre-Development, Development, and Post-Development.

The proposed lifecycle is based on the activities and processes required by DO-178B. It offers a flow involving all those activities whenever they are needed. This helps developers to do the right thing at the right time.

As discussed in Section 4.1 progressive and iterative types of lifecycle are not compatible with safety-critical development projects due to the special characteristics of the latter. The proposed lifecycle is sequential in type however it does not include a testing stage since as demonstrated in Chapter 3, testing safety-critical software is unfeasible. Testing is replaced by a system verification stage that verifies whether the end-product functions in accordance to requirements or not. System verification is further explained in the following section and a method is proposed as a tool to efficiently verify the developed software.

The weakness of the proposed lifecycle is in the possible occurrence of loops within it due to changes and modifications that are very familiar in software development projects. Those loops were grouped into the Plan-Build Release cycle as explained in Section 4.2.3.4. In Chapter 5 an SQA model is proposed. One of its objectives is to overcome loops and iterations within the proposed lifecycle. The effects of this SQA model on the proposed development lifecycle are discussed in Section 5.3.

4.3 System Verification

System verification is not testing. While Testing is running the software several times in samples in order to detect errors, Verification is assuring that a product corresponds with all functional requirements and specifications. As agreed by most regulatory bodies, safety-critical software can only be verified when fully integrated with their system mainly in order to verify their functionality and performance within the system. Verification is conducted on the system and software as a whole.

The objective is to check if all functions required from the designed software are functioning in accordance with the design requirements. Each function is to be verified for its own purpose. Hence, system verification is a functional verification method.

The approach is to deal with each function being verified as a "black box", meaning that the verification process is only interested in the function's output and not on how the output was achieved. Each "black box" has a known input and a desired output which is predefined by the software requirements and design. When verification is conducted, the actual output is compared with the desired output, and if they are both the same or close within a predefined allowable margin of error, then the verification result is positive. Figure 4.7 illustrates the "black box" approach.

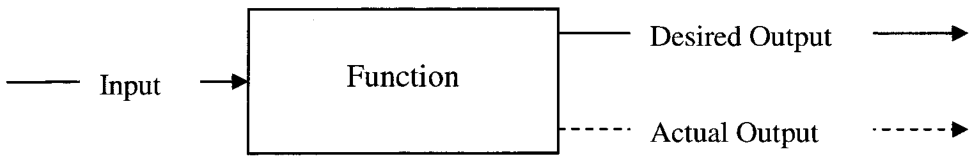


Figure 4.7: Black Box Functional Verification

In the case of a failure, errors are documented and the developer is required to review the failed function's design and code in order to fix it.

Software functions can be of two types: dependent or interdependent. The verification of each type differs in its complexity. The following section defines dependent and interdependent function types.

4.3.1 Independent and Interdependent Functions

Functions that run without using any other function from the same software being verified are called Independent Functions. Functions that use each other to run are called

Interdependent Functions. We can differentiate between two types of interdependency: "one-way" and "two-ways". A "one-way" interdependency is where a Function X depends on Function Y but without Y depending on X. In the case where X depends on Y and Y depends on X the interdependency is called "two-ways". Figure 4.8 is a graphical representation of One-Way and Two-Ways interdependencies.

Y
X
One-Way
Y
X
Two-Ways

Figure 4.8: One-Way and Two-Ways interdependencies

Contrary to Independent Functions, Verifying Interdependent Functions can be very complex. This is demonstrated in the following section.

4.3.2 Verification for Independent and Interdependent Functions

Verifying an Independent Function is a simple procedure. As mentioned earlier, the function is verified as a black box for whether it performs its intended role or not. In this case, a defect means that the error lies in the function itself: it could have been coded and/or designed wrongly. This cannot be applied to Interdependent Functions; a defect can be the result of an error in the function itself and/or in one or more of the functions upon which it depends. The more complex the function is in its dependencies with other functions, the more complicated the verification procedure gets.

The complexity of functional verification is directly related to the complexity of the software and the dependencies in its functions. In large software projects, this complexity can turn into a testing chaos and result in a waste of time and resources. In order to solve this problem, a "bottom-to-top" approach to functional verification is proposed.

4.3.3 Bottom-to-Top Functional Verification

No matter how complex the functions in a computer program are, they all can be rooted down to one or more independent function. The idea behind the proposed Bottom-to-Top Functional Verification (BTFV) is to sort all functions regarding their dependencies and to start the testing procedure from the independent functions up to the most complex functions.

The first step in BTFV is to build a Dependencies Tree which illustrates the dependency relations between all functions in a tree format composed of n levels where n=0 corresponds to the level of strictly independent functions (i.e. the root level of the tree). As the value of n increases, the complexity of the functions in their dependencies increases, and hence the complexity of the software. Therefore, n can be also referred to as the “dependency level”.

It is important to note that a function of level n cannot depend on any function of level $n+i$ where i is a positive integer. Moreover, a function of level n is necessarily dependent on a function of the level $n-i$.

The Verification sequence starts from Level 0 of the Dependencies Tree since all functions belonging to this level are independent and can be verified without relying on the result of any other function. At Levels n where n is greater or equal to one ($n \geq 1$), the verification test can start with any function of this level but it is preferred to start with the function that has the highest number of dependent functions; a defect in this function affects more other functions. In the case of a tie, it is broken randomly.

In the case of a defect, the test continues on the remaining functions of the same level while the defective function is sent back with its testing documentation to be fixed.

When all the functions of level n are tested positively, the test can move on the functions on level n+1 until all levels are tested.

Verification records are documented in the Software Verification Results document required by DO-178B. It should contain the following:

- Name and configuration of each function being verified.
- Result of the verification of each function.
- Reason(s) of failure (for functions that don't pass the verification test).

4.4 Chapter Summary

Software lifecycles are necessary to define and organize the flow of all development activities. Lifecycles can be sequential, iterative, or progressive. In this chapter, a lifecycle was proposed for safety-critical software projects based on DO-178B and experience acquired from the CRIAQ project. While the proposed lifecycle is of a sequential type, an iterative cycle, known as the Plan-Build-Release cycle, is found within in it. Each component of the software being produced follows its own Plan-Build-Release cycle which ends when the component is accepted after verification. Also, the complete software and its system should be verified as a whole before proceeding to Certification and Validation, the last activity of the project lifecycle.

Verification is necessary to guarantee that the produced software's operational functions fulfill their purpose and output within the tolerated error range the expected result. As safety-critical software products are large and complex, their verification gets extremely complicated. Functions are dependent on each other; the output of one may affect the output of several other functions. Therefore, a Bottom-to-Top Functional Verification approach was proposed in this chapter. It is based on functions' interdependencies. The main reasoning is to sort functions in a Dependencies Tree, then start by verifying functions whose output is not affected by a wrong output from another function, and move on up the tree until all functions are verified. By doing so, when a faulty output is found then the error is for sure in the function itself and not in a faulty input.

Despite validations made by experts and third party regulators, the software developer is the main person responsible and accountable for the product's safety. At the same time, the developer is highly interested in delivering the software on-time, within the budget, and with all requirements –especially safety requirements- satisfied, hence the Quality Challenge.

The challenge is to avoid iterating the Plan-Build-Release cycle and to have the software developed right and according to all requirements from the first iteration of the project lifecycle. Achieving this will save money, time, and resources for the developing party, not to mention other gains such as good reputation and reliability.

Ensuring and assessing the software product at each stage of the development lifecycle is the key path to achieve the Quality challenge. For that purpose, a Quality Assurance model is proposed in the next chapter. It consists of a set of acceptance criteria developed in accordance to the characteristics of safety-critical software. Applying this model concurrently with development stages throughout the proposed lifecycle ensures a continuous flow of the project without any iteration or cycle.

5.0 SQA MODELLING APPROACH

As discussed in Section 3.4.4, in commercial avionics systems, DO-178B and its European equivalent ED-12B are the main documents used as the means to ensure compliance with regulatory, certification, and safety requirements. DO-178B describes the objectives of each stage of the software lifecycle, the stage activities, and the evidence of compliance required. However, DO-178B does not set any norms or guidelines on when to consider a stage complete and the software ready to move to the next development stage.

In order to achieve the quality challenge, it is of high importance to ensure that each stage in the lifecycle is completed error free and with all requirements met accurately. Therefore, a software quality assurance model is developed in this chapter. It consists of a proposed set of quality acceptance criteria derived by the author. These criteria complete DO-178B by being indicators and rules according to which quality assessment is carried out after each development stage and the final conclusions about software conformity the requirements are done. The proposed set includes four criteria: Completeness, Documentation, Intelligibility, and Independence.

The first part of this chapter introduces and defines each of the four proposed criteria. Then, an explanation is presented on how the proposed SQA model is implemented within the lifecycle and how it affects it.

The second part of this chapter completes the proposed SQA model by providing a framework and a set of guidelines for each of the four criteria. The chapter is concluded by a diagram illustrating the proposed SQA model and its implementation.

5.1 Quality Acceptance Criteria

A set of four quality acceptance criteria forming the foundation of the SQA model are proposed in this thesis. The criteria were derived by the author from readings and analysis of certification and validation requirements set by regulatory bodies as well as from the experience drawn from the CRIAQ project. The criteria are proposed as an assessment tool to ensure that each stage of the development lifecycle of safety-critical software is consistent with DO-178B requirements. This will prevent iterations within the lifecycle and build quality into the software from as early as the first stage, hence favouring a certifiable product developed within budget, timeline, and in compliance to all requirements.

A successful and certifiable safety-critical software project is a project with all activities, tasks, and objectives fully complete and in conformance with planned requirements and norms. Moreover, all regulatory bodies require accurate documentation of the project development and outputs that are clear and traceable. They also emphasize the independence of personnel conducting tests, verifications, and assessments throughout the development. Therefore, the proposed set of acceptance criteria is made of the following four criteria that embrace all the characteristics of a successful and certifiable safety-critical software development project: Completeness, Documentation, Intelligibility, and Independence.

5.1.1 Completeness

The first quality acceptance criterion is Completeness. Besides errors and bugs, loops in the software lifecycle are also due to unfinished activities, unmet or forgotten requirements, and unachieved stages' specifications and objectives. The purpose of the Completeness criterion is to ensure that a development stage fulfills all its activities while accomplishing all goals and requirements. The aim is assuring that the next stage in the lifecycle can be launched with zero risk of having to go back to an earlier stage.

The Completeness criterion requires that the development stage being verified for quality acceptance satisfies the following set of norms of completeness:

- Specifications

All specifications developed at that stage match accurately and completely all the specifications required for the software project in question.

- Functional Requirements

All system and software functional requirements are present in the development progress at that stage. If a certain functional requirement cannot be worked on at the current stage, its exclusion should be justified.

- General Requirements

All general requirements such as standards and safety issues are an integral part of the development stage in question.

- Objectives and Goals

All objectives and goals of the current stage have been completely met and justified.

5.1.2 Documentation

Documents are an integral part of software development and of the software product (e.g. user manuals). No software can be certified and/or validated without the presentation of sets of documents required by regulatory bodies. For instance, DO-178B requires one or more document to be prepared at each development stage. Documents are also necessary for activities and information flow between stages; coding requires design documents, and design is based on software requirements documents. Therefore, it is important to make sure that all necessary documentation is completed on time; hence, the importance of the Documentation criterion.

The Documentation criterion is satisfied if all necessary and required documents from the development stage in question are completed according to standards, norms, and rules required for the project (e.g. DO-178B documents). All documents in question are to be finalized then audited and accepted by the Quality Assurance member(s) of the Developer's team. It is important to note that Completeness and Documentation are two interrelated criteria that go in parallel with each other.

5.1.3 Intelligibility

The Intelligibility criterion is made of two norms, Clearness and Traceability, which both should be satisfied in order to achieve Intelligibility.

- Clearness

All documents and all released material are to be clear and understandable to the client and third party experts who did not take part in the development process but are auditing the process. Consequently, Clearness allows faster and more efficient reviews, evaluations, and assessments.

- Traceability

Each stage should be traceable (linked) to its previous stages. Traceability should also be present between system requirements and software requirements, between low-level requirements and high-level requirements, and between the source code and the low-level requirements. Traceability keeps track of the flow and evolution of all development activities that facilitate the detection of development mistakes and sources of any error, bug, or defect.

5.1.4 Independence

The Independence criterion is related to all verification and audit processes. These processes should be carried out by an expert or group of experts who is/are independent from the developer's team. Verification, audit, and testing cannot be conducted by the same person or group who developed the software. Independence is relative; the verification experts can be a part of the same organization developing the software as they can be from a different organization administratively and/or financially independent. The more the software is safety-critical, the more independence is required between the developer and the verification expert(s).

5.2 The Use of Quality Acceptance Criteria

The proposed set of Quality Acceptance Criteria is designed to be applied after each development stage. However, due to the characteristics of each criterion, not all of them are applied to all of the development stages. Table 5.1 summarizes which criteria to apply at each development stage of the proposed lifecycle; a (+) sign means that the criterion has to be applied where a (-) sign means that it does not. A development stage cannot be started before the preceding stage has been found to satisfy all of the applicable criteria in the set.

Table 5.1: Acceptance criteria by stage

	Completeness	Documentation	Intelligibility	Independence
Project Initiation	-	+	+	-
Syst. Requirements	+	+	+	-
SW Planning	+	+	+	-
SW Requirements	+	+	+	-
SW Design	+	+	+	-
Coding	+	+	+	-
Code Testing	+	+	+	+
Integration	+	+	+	+
System Verification	+	+	+	+
Cert./Valid.	-	+	+	(-)[1]

1. Independence is not required for that stage since no actual certification processes are taking place.

5.3 Development Lifecycle with Quality Acceptance Criteria

Applying the set of Quality Acceptance Criteria results in a change in the development lifecycle proposed in Chapter 4. As it has been said earlier, each development stage should be checked for applicable criteria before the following stage is started. In case of an error, the developer should restart the stage in question to fix all found problems. Figure 5.1 illustrates the development flow when the set of Quality Acceptance Criteria is applied.

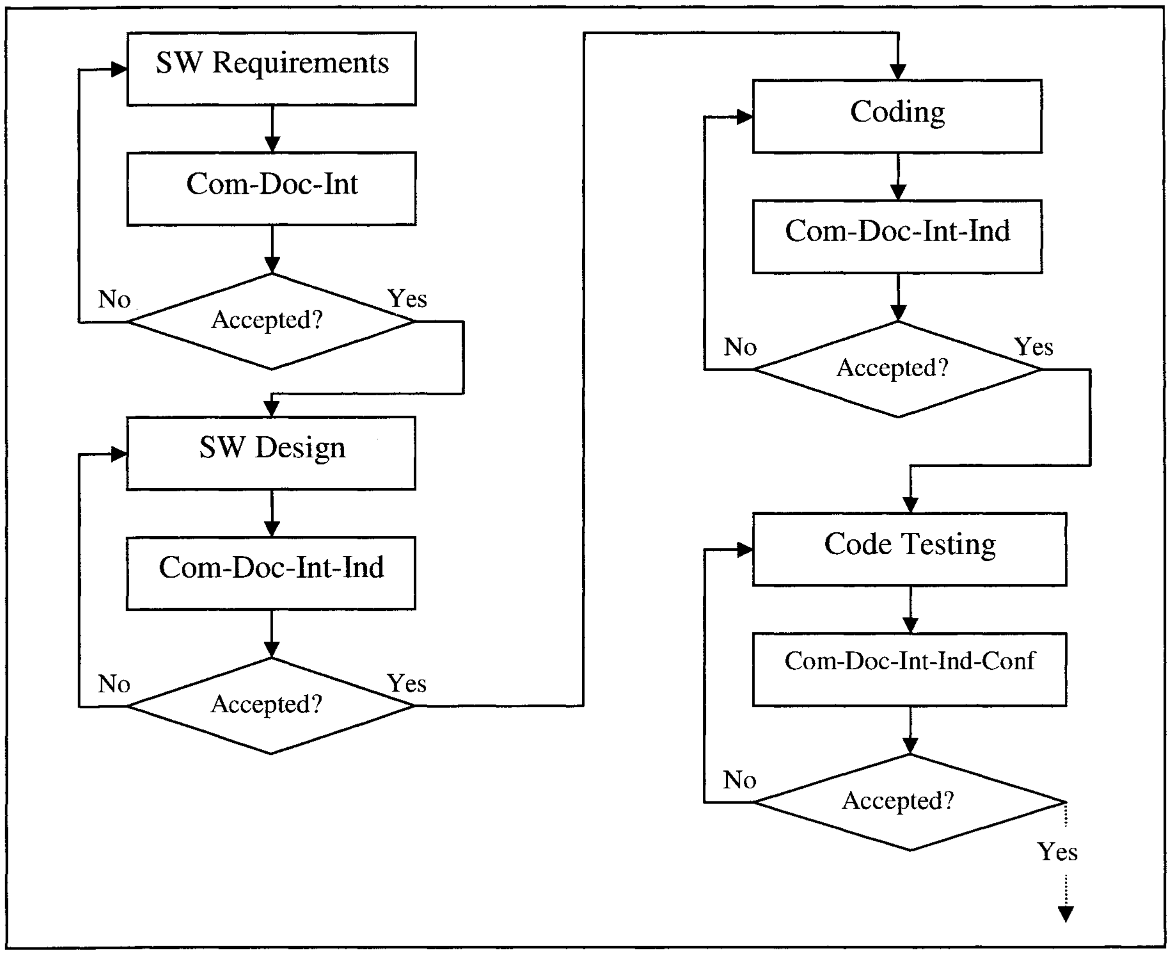


Figure 5.1: Quality Acceptance Criteria applied to the development lifecycle

Carrying out an assessment of the set of Quality Acceptance Criteria as stated in the previous paragraph will ensure to the developer that the development of the software is on the right track and that it will be completed right from the first development loop. However, this assessment after each stage has its drawbacks as well. It introduces an extra process right after each development stage which can be time and resource consuming; safety-critical software development project tend to go over schedule. In addition, it introduces a new loop to each stage as illustrated in Figure 5.1; this can be seen as breaking the Plan-Build-Release loop into four smaller loops.

In order to overcome these problems, the Quality Acceptance Criteria assessment should be applied concurrently with the development stage. While developers are working on the software development, a team of Quality Assurance works on evaluating the progress in the current stage with respect to the set of Quality Acceptance Criteria. Figure 5.2 illustrates the proposed development flow when concurrent assessment is applied.

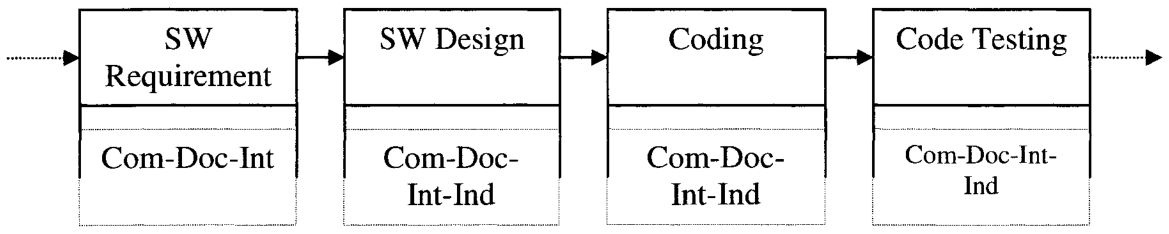


Figure 5.2: Concurrent assessment for Quality Acceptance Criteria

The advantage of this concurrent approach is time saving and the avoidance of having regressive assessment loops within each stage. The following discusses and proposes frameworks and guidelines for the application of each criterion in addition to a set of rules on how to accept or reject the assessment of a development stage with regard to each criterion.

5.4 Frameworks for the Proposed SQA model

In the above, four Quality Assurance Criteria were proposed: Completeness, Documentation, Intelligibility, and Independence. An emphasis was made on assessing each stage using these criteria –when applicable- concurrently with the development activities in order to ensure a neat development flow and avoid iterative loops, repetition, and waste of time and resources. Enforcing the Quality Assurance Criterion is the responsibility of the SQA team working on the software project.

For each criterion, this chapter proposes a framework developed by the author describing how the SQA team and the Development team can, while working together, ensure that each stage will be accepted from the first Quality Assurance Criteria assessment. Acceptance and rejection are based on a set of rules that are also proposed in this chapter for each criterion.

5.4.1 Framework for Completeness

As discussed in Section 5.1.1, the purpose of the Completeness criterion is to ensure that the development stage being assessed fulfills all its activities while accomplishing all goals and requirements. Mainly, Completeness is related to Requirements (both

Functional and General), and Objectives and Goals of a stage. The general rule is that a stage is considered "complete" only if all Requirements, Specifications, Objectives and Goals are fulfilled.

In the following, the author proposes a framework aimed at ensuring Completeness for all development stages. In addition, Section 5.4.1.3 proposes a set of rules to be used when assessing a stage for the Completeness Criterion.

The proposed framework is developed for both aspects related to Completeness: Requirements, and Objectives and Goals.

5.4.1.1 Requirements

As illustrated in Chapter 4, Requirements are the backbone of the software development project. The System Requirements are defined first and then broken down into High-Level and Low-level requirements throughout the life-cycle. Whenever requirements are stated or defined, whether they are system requirements, high-level or low-level requirements, the SQA team is responsible for assessing them with respect to the severity of the consequences that might result from failing to meet each requirement or from a defect in operational functions related to each requirement. The following describes how this assessment is done.

5.4.1.1.1 Assessing Requirements

For safety-critical software for avionics and airborne applications, the assessment is made on a scale of 5 levels (A, B, C, D, and E) provided by DO-178B. Those levels are:

- **Level A - Catastrophic**

Requirements in which failure would prevent continuous safe flight and landing are of Level A.

- **Level B – Hazardous**

Requirements are of Level B if their failure would not prevent a continuous flight and landing but would result in a reduced capability of the aircraft or poor ability of the crew to cope with adverse operating conditions to the extent that there would be one or more of the following:

1. "A large reduction in safety margins or functional capabilities,
2. Physical distress or higher workload such that the flight crew could not be relied on to perform their tasks accurately or completely, or
3. Adverse effects on occupants including serious or potentially fatal injuries to a small number of those occupants".

- **Level C – Major**

Requirements are of Level C if their failure reduces the capability of the aircraft or of the crew to deal with difficult operating conditions to the extent that there would be a major increase in crew workload or in conditions impairing crew efficiency, or discomfort to occupants, possibly including injuries.

- **Level D – Minor**

Requirements are of Level D if their failure not significantly reduce aircraft safety, and would require crew actions that are well within their capabilities. Minor failure conditions may include a slight reduction in safety margins or functional capabilities, a slight increase in crew workload, such as, routine flight plan changes, or some inconvenience to occupants, without any injury.

- **Level E – No effect**

Requirements which failure do not affect the operational capability of the aircraft or increase crew workload are of Level E.

All assessment results are documented in their corresponding entry in the Requirements List Document, a template example of which is provided in Appendix B.

During the development stage, when the development team finishes work on a certain requirement (or a set of requirements), the team asks the SQA team to check if the requirement is met. The following sub-section describes the process of Checking Requirements.

5.4.1.1.2 Checking Requirements

The purpose of the Checking Requirements procedure is to make sure that all the stage's requirements are met. The Requirements List corresponding to that particular stage is used as a checklist in the verification process.

As mentioned earlier, requirements are branched down from System to Low-Level requirements. The process of checking requirements starts from Low-Level branches up to the main System Requirement. A High-Level requirement cannot be accepted as "complete" unless its objectives are met and all the Low-Level requirements derived from it are "completed". Correspondingly, a System Requirement is "complete" only when its objectives are met and when all of its High-Level requirements are "complete". This process is carried throughout the development stage at hand.

The second part of the framework for Completeness is to ensure that the stage meets all Objectives and Goals. The following section is dedicated to that purpose.

5.4.1.2 Objectives and Goals

Objectives and Goals can be translated into tasks to be performed during the development stage. In order to make sure that all those tasks are done, they should first be scheduled and planned. Each time a task is over, the SQA team is responsible for checking whether it is finished or not. The following two subsections propose a model to schedule tasks and a procedure to check tasks.

5.4.1.2.1 Scheduling Tasks

When tasks are derived and defined, they are first graded according to their importance. A scale of 4 Levels of Importance is proposed as follows:

- **Level 4: Crucial**

A task is graded 4, or Crucial, if it is one of the main tasks of the development stage in question and cannot be performed except during that stage, and if the next stage cannot be initiated without its completion.

- **Level 3: Very Important**

A task is of level 3, or Very Important, if it is a main task of the development stage in question and cannot be performed except during that stage, and if some of the tasks of the next stage cannot be initiated without its completion.

- **Level 2: Important**

Level 2, Important, tasks are those that have no effect on the next stages but some tasks of the current stage cannot be performed without their completion.

- **Level 1: Not Important**

Tasks of level 1, or Not Important, are those with no effect on the work progress of the current stage and do not have any effect on the following stage.

All tasks should be documented in a Tasks List document that includes the task's name or label, Level of Importance, and objective(s). Appendix C provides a template example of a Tasks List.

Once all tasks are levelled, they are ready to be scheduled. Using a Gantt chart, tasks can be scheduled throughout the time allocated by the Development Plan (Section 4.2.1.3) for the particular stage.

Usually, the first task to be performed is the first one to appear on the Gantt chart. However, several tasks can be scheduled to start at the same time (if precedence requirements allow it), and if resources are not available for all of them, a choice should be made to decide which task has a higher priority. The following method is developed by the author as a tool to decide on priority:

Tasks of levels 4, 3, 2, and 1 are given an arbitrary score of 10, 5, 2, and 1 respectively. A task's Total Priority Score (TPS) is computed by adding its own score to the score allocated to all tasks scheduled after it and cannot be started without its completion.

Consequently, when a choice is to be made in order to decide the allocation of available resources, the TPS of all tasks that can be started at that particular point in time and the TPS of all tasks left unstarted from previous resource allocations are to be computed and resources are allocated to the task with the highest TPS. If it is feasible to allocate resources to more than one task, the allocation process is done starting with the task with the highest TPS, followed by the one with the second highest TPS, and so on until all resources are allocated.

In order to illustrate this process, consider the tasks scheduled in the Gantt chart given in Figure 5.3; where the number given between parentheses corresponds to the Importance Level of each task.

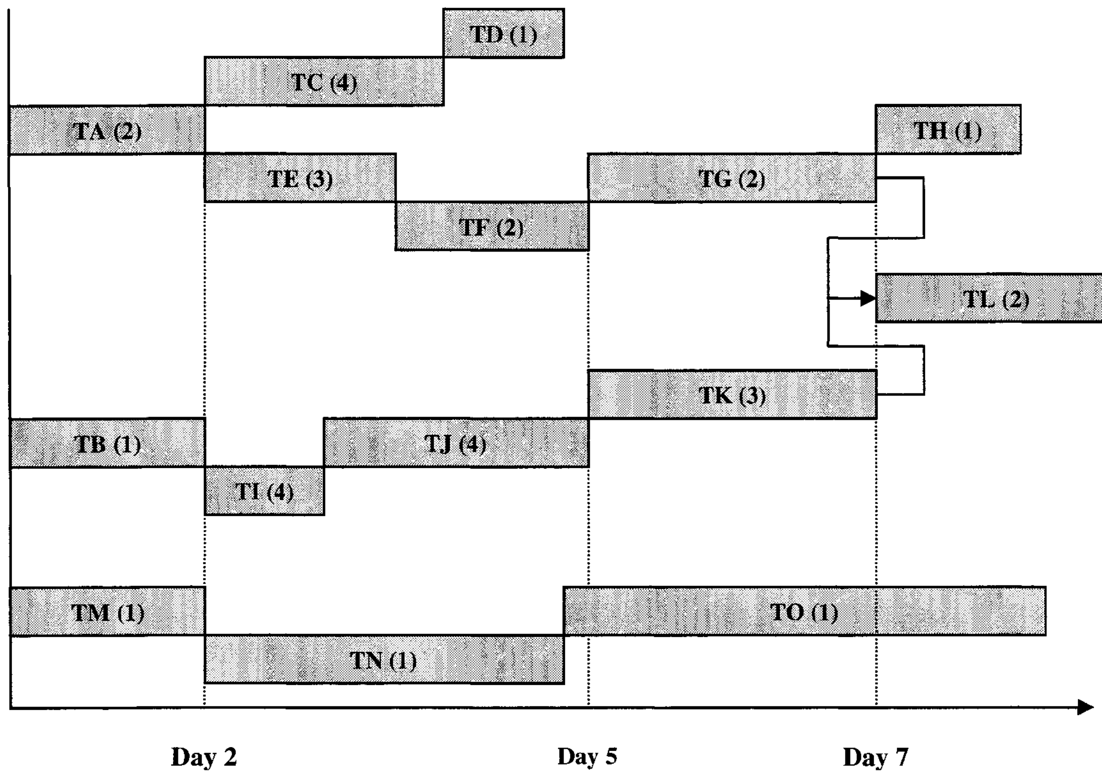


Figure 5.3: Gantt chart example

- The Total Priority Score of TA is:

TPS(TA)=TA+TC+TD+TE+TF+TG+TH+TL=2+10+1+5+2+2+1+2=25

(TA is of level 2 then TA=2 according to proposed scoring scheme per importance level. Similarly, TC is of level 4 then TC=10)

- The Total Priority Score of TB is:

TPS(TB)=TB+TI+TJ+TK+TL=1+10+10+5+2=28

- The Total Priority Score of TM is:

TPS(TM)=TM+TN+TO=1+1+1=3

Hence, if resources are only available for 1 task, then they are allocated to TB. If they are available for two tasks, then both TA and TB can start. Now, consider that a decision is to be made on Day 5 of the schedule and assume that TM has not been started yet. In this case, the TPS of TG, TK, and TM are computed and compared:

TPS(TG)=TG+TH+TL=2+2+1=5

TPS(TK)=TK+TL=5+2=7

TPS(TM)=TH+TN+TO=1+1+1=3

Therefore, TK goes first if resources can only be allocated to one task.

The Importance Level is also used when deciding if a stage can be accepted as Complete before moving to the following stage according to the rules of acceptance that are proposed in Section 5.4.1.3. The following describes the process of checking tasks by the SQA team.

5.4.1.2.2 Checking Tasks

The purpose of this procedure is to ensure that all tasks have been completed. It is performed throughout the development stage at hand. Whenever the development team finishes working on a certain task, the SQA team verifies if the task is completed or not. A task can only be complete if all of its objectives are met. If a major correction or adjustment was found to be necessary it should be noted in the Remarks section of the Task List. Once a task is judged complete, its corresponding "Is Complete" entry in the Task List is checked, and only then the subsequent task can start. Hence, the Task List is used as a checklist.

5.4.1.3 Rules of Completeness Acceptance

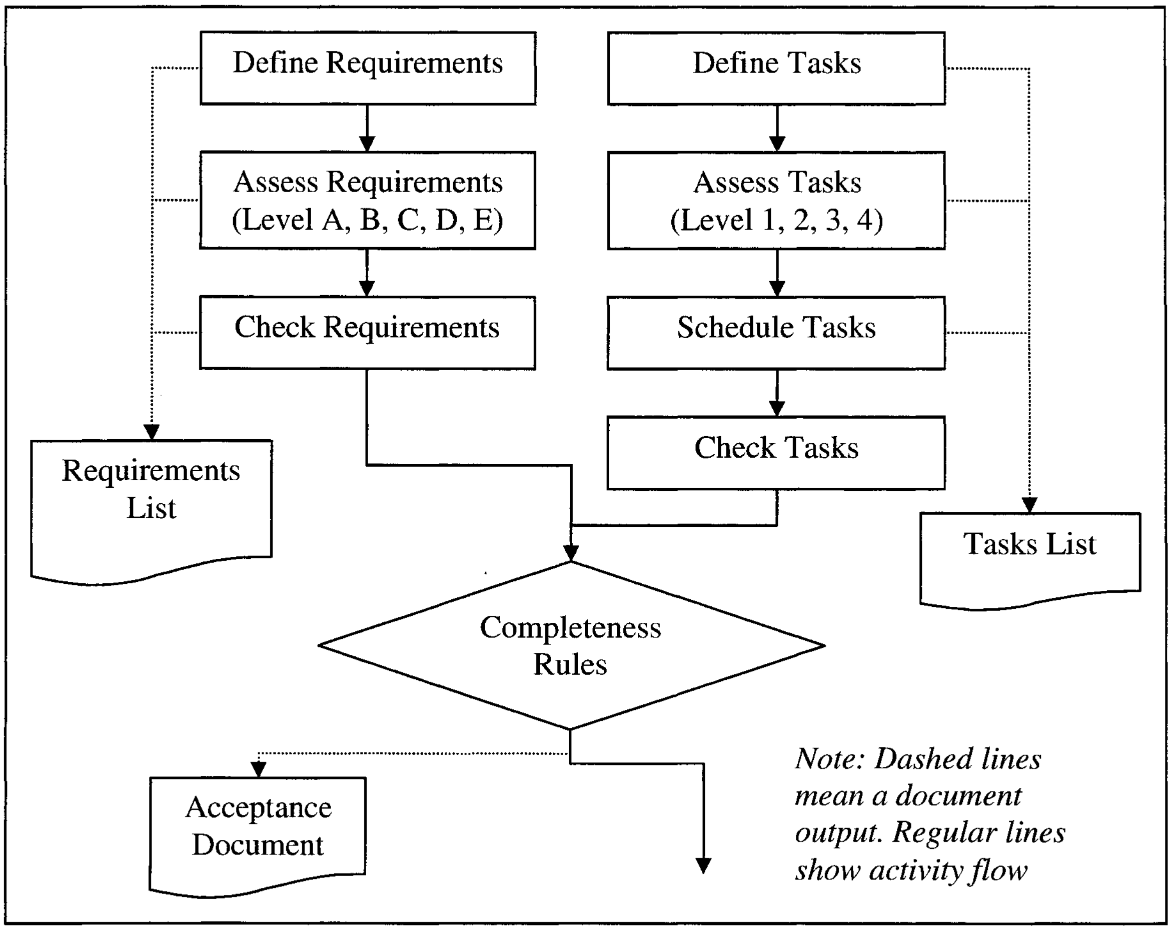
At the end of each stage, the SQA team compiles the results obtained from checking requirements and checking tasks to verify whether the stage in question can be accepted as Complete or not. As mentioned before, the general rule is that all requirements and all tasks should be met. The "firmness" nature of that rule is required since safety is critical, however, it delays the development process: some parts of the following stage can be initialized disregarding if all current requirements or tasks are met. Furthermore, some of the completed requirements and tasks can trigger activities in the following stage. Therefore, the following set of rules is proposed with a purpose to emphasize safety issues while keeping a flexible and delay-free life-cycle:

1. Tasks of Level 4 and 3 should all be completed
2. Tasks of Level 2 and 1 can be left incomplete
3. Tasks of Level 2 left from the previous stage should be completed
4. Tasks of Level 1 left from the stage preceding the previous stage should be completed
5. Requirements of Level A and B should be met
6. Requirements of Level C and D can be kept in progress if unmet
7. Requirements derived from unmet Level C requirements from the previous stage cannot be worked on
8. Requirements of Level C left from the previous stage should be met
9. Requirements derived from unmet Level D requirements from the previous stage can be worked on only if they are of Level D or E

10. Requirements of Level E left unmet from the stage preceding the previous stage should be met

These rules of acceptance apply to all development stages except for Certification/Validation, which is the last one of the development lifecycle. At this point of the lifecycle, all requirements and tasks relevant to the Certification/Validation stage and all those left from previous stages should be complete.

The Framework for Completeness is summarized in Figure 5.4 as a diagram illustrating the steps to be undertaken throughout the development stages where Completeness is a required quality acceptance criterion.



Define Requirements
Define Tasks
Assess Requirements (Level A, B, C, D, E)
Assess Tasks (Level 1, 2, 3, 4)
Check Requirements
Schedule Tasks
Check Tasks
Requirements List
Tasks List
Completeness Rules
Acceptance Document
Note: Dashed lines mean a document output. Regular lines show activity flow

Figure 5.4: Completeness framework diagram

5.4.2 Framework for Documentation

Documents are an essential part of the software development process. The goal of the Documentation criterion is to make sure that all documents are produced when they should be.

The first step towards ensuring Documentation is to identify all documents required as an output of the development stage under assessment. Then, all those documents are given a level grade according to their importance on the flow of the development lifecycle. A List of Documents is required for each stage listing all necessary documents with their level of importance a brief description. Appendix D shows an example of a template of a List of Documents. For documents, the following importance scaling system is proposed:

- Level 3: Very Important
- Level 2: Important
- Level 1: Normal Importance

The second step is to schedule all the documents throughout the stages timeline. The documents should be related to a stage's tasks in order to know when they can start. By doing so, a schedule for the documents can be drawn in parallel to the tasks' schedule and illustrated on the tasks' Gantt chart.

The general rule is that no development stage can be started unless all documents from the previous stage are done. However, sometimes documents can take a long time to be finalized as they require several reviews and many corrections can be necessary; this can cause delays in the development process. Therefore, some tolerance is needed to ensure a neat development flow. The following set of rules is proposed when assessing a stage for the Documentation criterion:

1. All Level 3 documents should have been reviewed at least once (i.e. two versions written).
2. All Level 2 documents should have at least a first version.
3. All Level 1 documents should have been started but not necessarily finished.
4. All Level 1 documents from the pervious stage should have been reviewed at least once.
5. All Level 1 and 2 documents left from the previous stage should be finalized.

Here again, these rules of acceptance apply to all development stages except for Certification/Validation. All documents relevant to the Certification/Validation stage and all those left from previous stages should be complete. Figure 5.5 is a diagram illustrating the framework for Documentation.

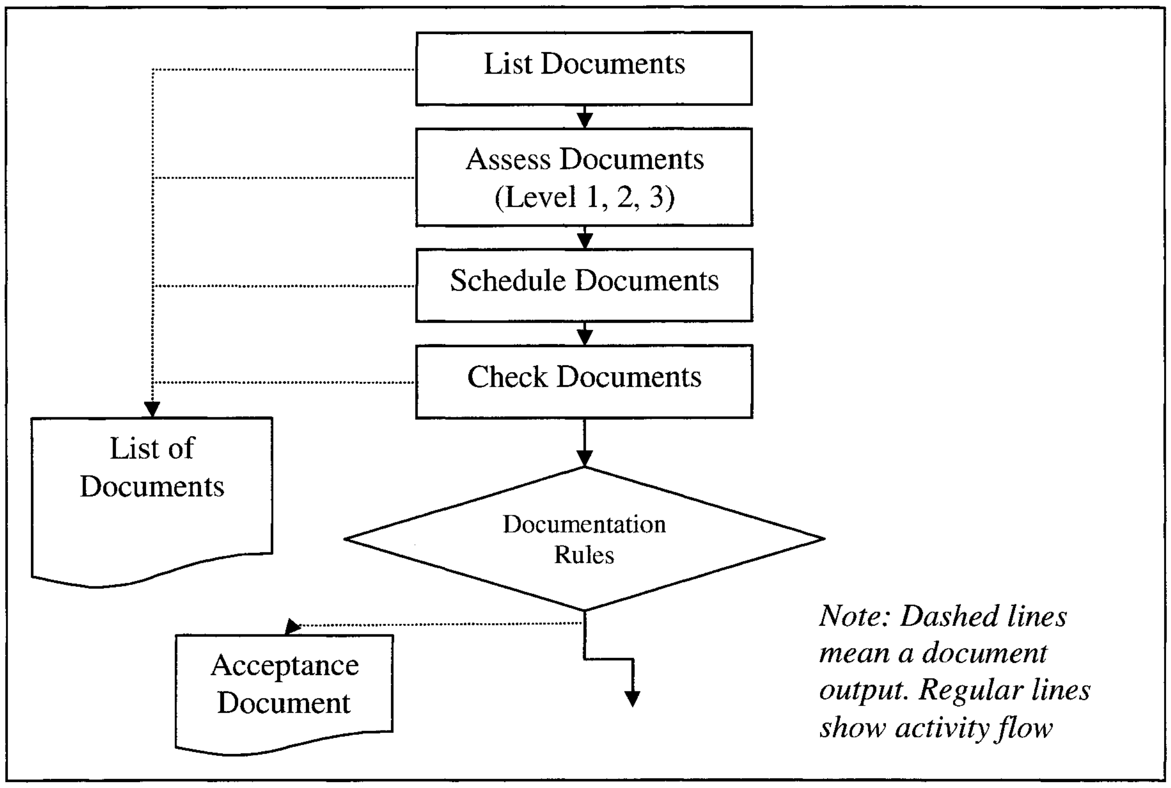

List Documents
Assess Documents (Level 1, 2, 3)
Schedule Documents
Check Documents
List of Documents
Documentation Rules
Acceptance Document
Note: Dashed lines mean a document output. Regular lines show activity flow

Figure 5.5: Documentation framework diagram

5.4.3 Framework for Intelligibility

As discussed in Section 5.1.3, a development stage is considered intelligible if the two norms Clearness and Traceability are satisfied. Hence, the framework for the Intelligibility criterion can be split into two "sub-frameworks" detailed in the two following sections: Framework for Clearness, and Framework for Traceability. In the last section, the rule to accept a stage as Intelligible is defined.

5.4.3.1 Framework for Clearness

Clearness is related to documents and all other material released during the stage under assessment. In order to achieve Clearness the following rules are proposed:

1. All documents should contain a "mission statement" explaining the objectives and purpose of the document, as well as how and when it can be used.
2. Every document should be a whole by itself; documents with different objectives and purposes should not be combined to form one document.
3. All technical words and abbreviations should be defined and clarified.
4. All figures, equations, and tables should be numbered and titled.

All documents should be assessed for those rules. All rules should be satisfied. When a document is found to be Clear, the "Is Clear" entry in the List of Documents is checked.

5.4.3.2 Framework for Traceability

Traceability is essential to keep the whole software process development linked together from one stage to another. Traceability mainly deals with three aspects of the development process: requirements, functions, documents and code. The following sub-sections propose guidelines to ensure their traceability.

5.4.3.2.1 Requirements Traceability

As already mentioned, software High-level requirements are derived from the system requirements; then, Low-Level requirements are obtained by breaking down High-Level ones.

It is important to relate all requirements to each other in order to maintain traceability. The relation is defined by keeping track of the generation of each requirement. All relations are noted in the Requirements List Document.

In the following, a labelling system is proposed to help relate all requirements through labelling each requirement in relation to its "genealogy":

1. System Requirements are labelled with an "S" followed by a number, for Example, S1, S2, and S3.
2. High-Level Requirements are labelled with an "H" followed by a number. Then, the label is followed by the label of the System requirement from which the High-Level requirement was derived. For example, the first High-Level requirement derived from S2, will be labelled: H1-S2
3. Accordingly, Low-Level requirements are labelled with an "L" followed by its "ancestors" requirements. For example, the third Low-Level requirement derived from H1-S2 is labelled: L3-H1-S2

Note: The severity level of the requirement can also be included in the label between parentheses after the requirement number. Examples: L3(E)-H1(D)-S1(B), or L4(A)-H2(B)-S3(A).

5.4.3.2.2 Functions Traceability

This activity is undertaken during the Design stage where the Development Team is building the software architecture and defining all needed operational functions based on all requirements at hand. Functions are labelled with an identifier followed by the label of the requirement they have been drawn from. The identifier can be "F" with a number.

During this activity, the SQA team and the Development Team should determine the expected output(s) from each function. Where it is feasible, acceptable margins of error are to be defined for each function's output(s).

All functions are compiled with their labels, expected output(s), and margins of errors in a document called Functions List. An example of a template of a Functions List is given in Appendix E.

5.4.3.2.3 Documents and Code Traceability

Documents and Code are affected by several changes happening during the development process. In order to maintain their traceability, the following guidelines are proposed:

1. All changes, corrections, and modifications are made in a new version. Old versions are kept.
2. All documents and released code should contain a "document number", version number, and date of release.
3. All previous versions should be listed with a brief explanation of what has changed from a version to the next.

5.4.3.3 Rule for Intelligibility Acceptance

The rule for accepting a stage as Intelligible is that all its outputs are Clear and Traceable. Since safety-critical software projects are large and involve a large number of documents, code parts, objectives, requirements, and functions that are transferable from a stage to the next, none can be left as unclear and untraceable from a stage to another.

The framework for the Intelligibility quality acceptance criterion is illustrated in Figure 5.6.

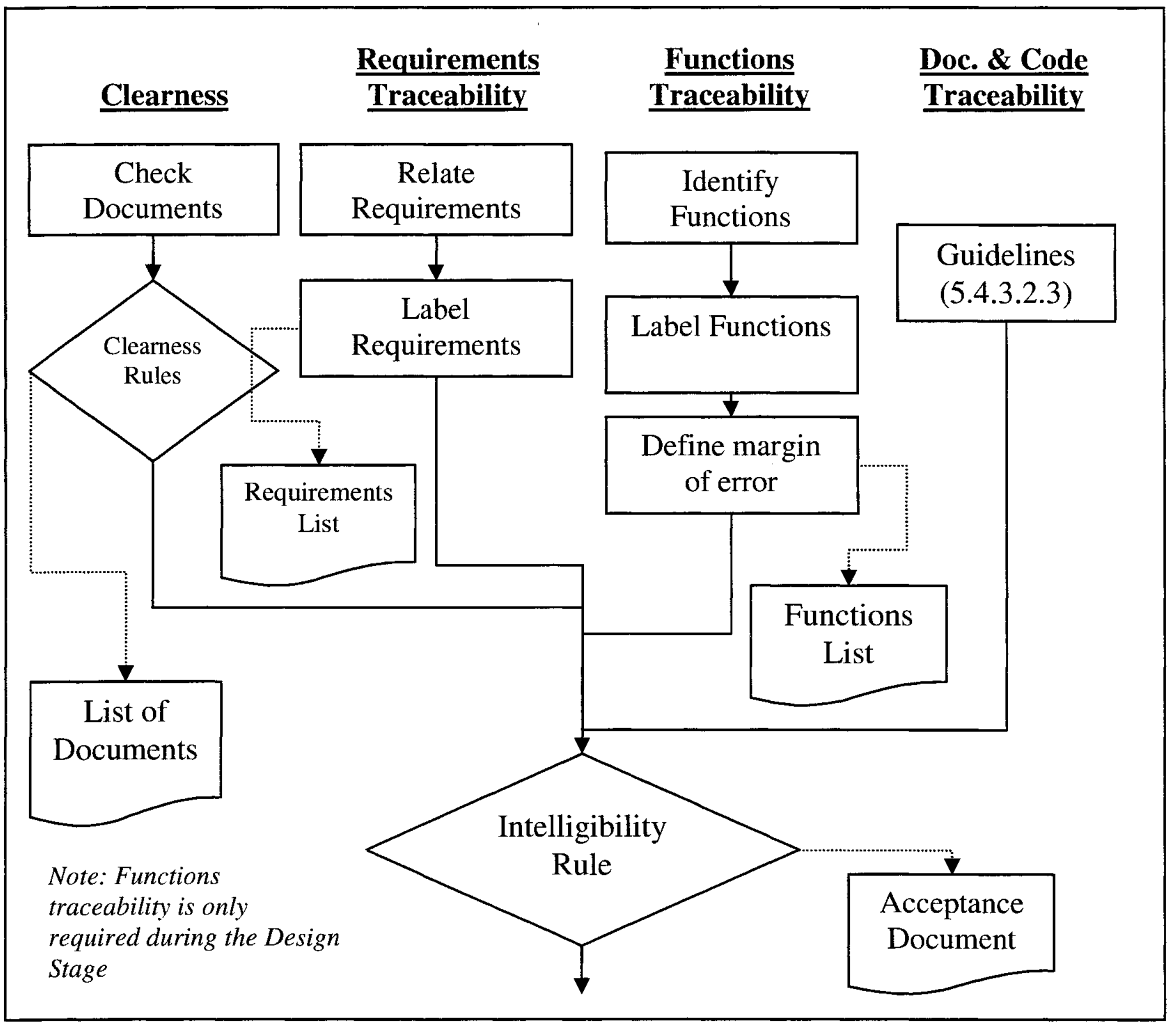


Figure 5.6: Intelligibility framework diagram

5.4.3 Framework for Independence

The purpose of the Independence criterion is to ensure unbiased assessment, verification, evaluation, and audit processes. A stage can be accepted as Independent if only both of the following rules proposed by the author are satisfied:

1. Personnel performing evaluation and verification processes have not been involved in the development process of the software.
2. Personnel performing evaluation and verification processes have not established the criteria, standards, and norms against which the project is being gauged.

Independence also reduces errors resulting from oversight due to extensive familiarity with the product being evaluated. Personnel performing independence related activities can be either an internal team (from the same organization developing the software but did not take part in the development) or an external team (from outside agencies and organization not related to the development).

The second rule for Independence acceptance is achieved by having one team (either internal or external) do specific assessments, evaluation, audits, and verifications, but a different team (either internal or external) establishes criteria and norms. Here again, both teams should be absolutely independent from each other.

5.4.4 Frameworks and the development lifecycle

So far in this thesis, a development lifecycle and an SQA model based on a set of four acceptance criteria were proposed by the author; their aim is developing safety-critical software projects meeting all extremely high quality requirements through an iteration-free lifecycle within budget and timeline. Moreover, for each proposed acceptance criterion a framework was designed with rules defining how a development stage can be accepted for that particular criterion. The following describes how the lifecycle and the SQA model interact together forming one method for safety-critical software development in parallel with DO-178B.

There are two teams involved during the development of software: the development team and the SQA team. Throughout the lifecycle, while the development team works on its own activities, the SQA team works in parallel building quality into the product from the early stages. Development teams tend to be more focused on building the software, hence the need for the SQA team to monitor quality and ensure it. Communication and coordination are a key factor between both teams.

During each development stage, the SQA team work on ensuring that the stage will be assessed positively for all acceptance criteria that apply. This is done by following the frameworks designed earlier in this chapter. Whenever a stage passes the assessment of a criterion it is noted so in the Acceptance Document a template example of which is given in Appendix F. As illustrated in Figure 5.7, a transition from a stage to the subsequent is only possible when the former passes the assessment which is based on the acceptance rules developed earlier for each criterion.

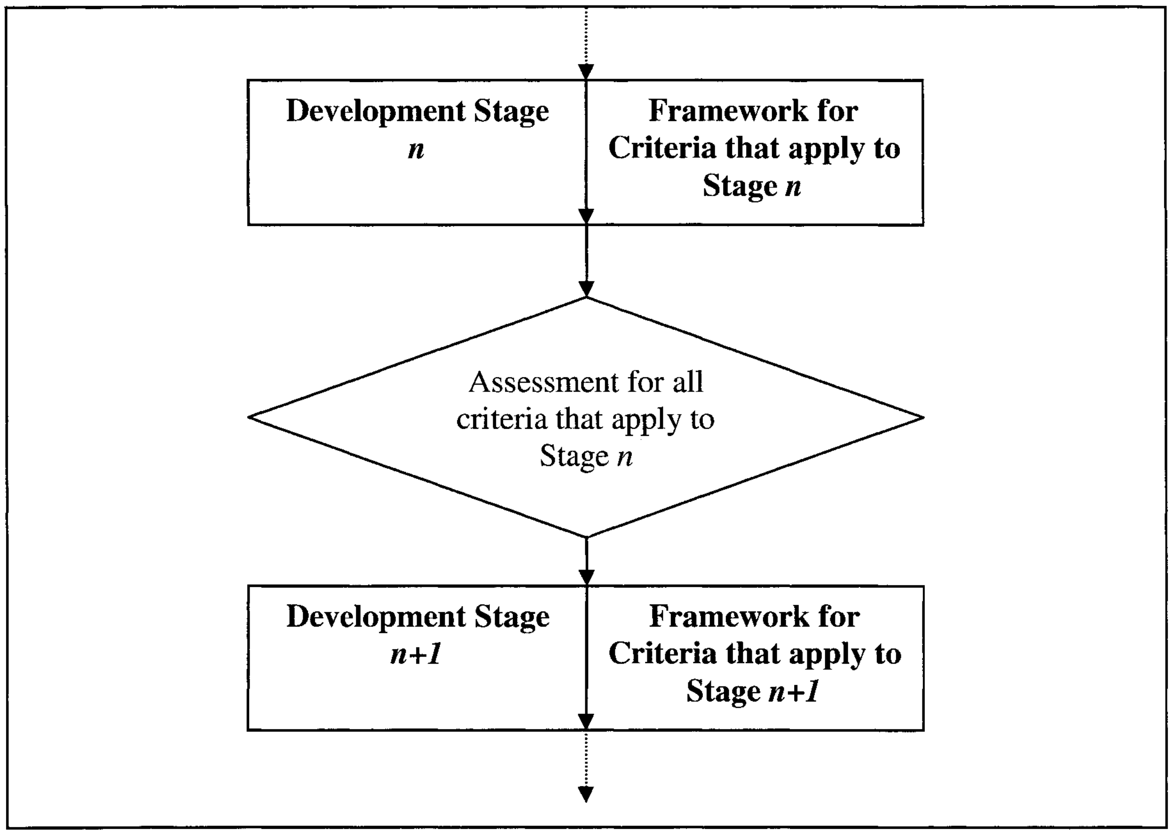

Development Stage *n*
Framework for Criteria that apply to Stage *n*
Assessment for all criteria that apply to Stage *n*
Development Stage *n+1*
Framework for Criteria that apply to Stage *n+1*

Figure 5.7: Transition from a development stage to the next

Table 5.2 summarizes the interaction between the lifecycle and the SQA acceptance criteria from the initiation of a safety-critical software project until the end. For each development stage the table shows which criteria apply, what output documents are required according to DO-178B and/or according to the SQA model designed in this thesis. When the proposed framework for a certain criterion is applied, certain documents should be outputted or updated; the last five columns of Table 5.2 show which of those documents are required and at each development stage.

Table 5.2: Lifecycle and SQA

Stage	Acceptance Criteria				Document Output		Proposed framework output documents				
	Completeness	Documentation	Intelligibility	Independence	Do-178B	Thesis	Requirements List	Tasks List	List of Documents	Functions List	Acceptance Doc.
Project Initiation		√	√			-Project Contract			√		√
System Requirements	√	√	√			-Definition of requirements	√	√	√		
SW Planning	√	√	√		-SW development plan - SW verification plan -SCM plan -SQA plan -Plan for SW aspects of certification		√	√	√		√
SW requirements	√	√	√			-SW requirements document (High-level)	√	√	√		√
SW Design	√	√	√		-Design Description	-SW requirements document (low-Level)	√	√	√	√	√

Coding	√	√	√		-Source code		√	√	√		√
Code Testing	√	√	√	√		-Test results and recommendatio ns	√	√	√	√	√
Integration	√	√	√	√		-Integration Record	√	√	√		√
System Verification	√	√	√	√	-Verification results document		√	√	√	√	√
Certification/Validation		√	√			-Checklist of documents required for certification	√	√	√	√	√

5.5 Discussion

When applied to safety-critical software development projects, the proposed SQA model ensures that the end-product is of high-quality by keeping track of all activities throughout the development and making sure that nothing is missed. It assists the developers in building quality into their product right from the initiation of the project.

By assessing each development stage for four acceptance criteria before allowing the next stage to start, the proposed SQA model does not tolerate any error/defect to be transmitted or inherited from a development stage to a subsequent one. This strictness is the main difference between the proposed model and other models (such as CMM and SPICE) which are leaner on defects and errors.

From the managerial side, the advantage of the proposed SQA model is that it starts by requiring all activities and documents to be well defined at the beginning and then follows rigorously the progress of each activity and document until its completion while ensuring that none is left undone. The SQA model also stresses on the traceability and clearness of all the work performed throughout the development. Moreover it emphasises on independency of verifications and assessments when they are required.

For software in aerospace applications DO-178B is the standard used for developing a certifiable product. DO-178B is at the heart of the framework of the proposed SQA model which when applied assists developers to perform an activity or output a document at the right time as required by DO-178B. In addition, the proposed model complements DO-178B by offering a method to assess all development lifecycle stages with respect to their corresponding DO-178B activities, actions, and requirements.

The proposed work in this thesis was based on the development of the CRIAQ project. This development followed only DO-178B guidelines. DO-178B does not offer any means to verify that all of its requirements are being met, and several activities of the CRIAQ project fell short. This inspired the proposed SQA model. For instance one of the software components developed had to be redesigned and rebuilt several times because of requirements that have not been met and because of requirements that just have been forgotten. In the proposed model, all requirements should be clearly defined and each stage is assessed on whether all of those requirements are in the development or not.

From the documentation side, some documents were still incomplete as the project progressed through time. This was mainly due to the absence of a definition and description of all documents required and of a time limit for their completeness. In the proposed SQA model, all documents should be clearly described at the beginning of each stage and given an importance level. According to the documentation criterion acceptance rules this importance level sets a certain time limit for each document; thus, at any time in the development, activities cannot be undertaken unless all documents due at that time are complete.

In addition, several software components and documents were left without any configuration what fails to meet traceability requirements which are essential in DO-178B and the proposed model.

At the time of completion of this thesis none of the developed components were verified for operational functions and none of the tolerances were properly defined. If the proposed SQA model was applied to the CRIAQ project all those components would have been verified by now since the model does not allow the progress of the development without a successful verification of all work done.

6.0 CONCLUSION

6.1 Summary

Software applications in which failure may result in possible catastrophic consequences on human life are classified as safety-critical. These applications are widely used in a variety of fields and systems such as airborne systems, nuclear reactors' control, medical monitoring and diagnostic equipment, and electronic switching of passenger railways. Unfortunately the world has seen several accidents and tragedies caused by failures of safety-critical software.

The quality of software depends on its developing system and methods. While the industry is aware of the importance of high quality and reliability of safety-critical software, very little research has been done on the quality of such products and their developing systems.

This thesis looks into safety-critical software embedded in systems in the aerospace field. Those systems can either be airborne or ground-based systems (i.e. aircraft simulators). The following contributions have been made:

- Demonstration that methods used for quality control and management of software and products in general cannot be applied effectively to safety-critical software products.

- Proposal of a lifecycle specially modeled for the development of safety-critical software in aerospace and in compliance with the DO-178B standard.
- Proposed of an SQA model that builds quality into safety-critical software throughout its development.

Quality control techniques as used in general rely on analyzing data and statistics collected from testing samples or all products produced. In safety-critical software the tolerance on defects is extremely small; only products with a defective rate smaller than 10^{-8} can be accepted which makes testing safety-critical software for errors infeasible due to the time it would take to detect an error. Hence quality should be built into safety-critical software and managed throughout the development.

A development lifecycle was proposed in this thesis. It models all development activities from the project initiation until the end product's release into a sequential flow of ten stages. Since testing is not feasible, verification replaces it. Verification does not look for errors, it verifies if the product performs its operational functionalities or not. One of the advantages of the proposed lifecycle is that it clarifies the flow of activities described in DO-178B. Moreover, it assists in the implementation of an SQA method by showing clearly the path the development is following, hence quality engineers would know what to do exactly at each point in the lifecycle.

This thesis proposes an SQA model designed for safety-critical software embedded in systems in the aerospace field. The model was designed by the author based on the DO-178B standard and experience and lessons learned while working as an SQA engineer on a project to develop a software components for an aircraft simulator that will be used as a Test-Bed for Flight Management Systems. While the proposed model can be implemented to all types of safety-critical software projects, it was mainly designed for "make-to-order" development projects which in general are very specific and clear.

This SQA model is based on assessing each development stage for four acceptance criteria: Completeness, Documentation, Intelligibility, and Independence. Each development stage should satisfy the rules of acceptance of all the criteria that applies to it (for some stages only the first three criteria are required, for all others the four are necessary). A development stage cannot be initiated unless its preceding one was accepted. Completeness looks into whether all the stage's requirements, activities, and tasks were met, completed, and performed according to set norms. The aim of the Documentation criterion is to ensure that all documents required from the stage being assessed were outputted and ready. Intelligibility requires that all outputs (documents, code, etc) be clear and traceable. Clear means that the output can be easily understood from people who did not take part in the development. Traceability is extremely important to link all outputs and activities together so that whenever a change is required it could be done easily and effectively. Independence requires that whenever an assessment and/or verification are required, they should be performed by an unbiased person or a team who never took part in the development.

While developers are working on their activities following the proposed lifecycle, an SQA team is responsible of implementing concurrently the proposed SQA model during all stages. The SQA team should be independent from the developers; however, excellent communication between both of them is the key for success. Frameworks for the implementation and assessment of each of the four criteria were also proposed. In addition, a framework showing how the SQA model should be executed through the lifecycle was proposed.

The work proposed in this thesis does not replace DO-178B; it however complements it by offering methods ensuring its implementation appropriately. Several developers in the industry have complained that DO-178B is not clear enough and does not include any method to verify that it is being accurately applied during the development. The lifecycle proposed in this thesis offers a clear flow of all the activities required during development, while the proposed SQA model checks each development stage for its compliance with DO-178B and all other requirements set by the developers.

6.2 Future Work

Safety-critical software is an emerging field of research. This thesis proposed a new model and approach to safety-critical software quality assurance. Like all new contributions in a field that is on a continuous technological advancement, it opens new doors for future research, expansion, reviews, and adjustments. The following presents some ideas for future work:

- Develop metrics to measure during a development project the implementation of the proposed SQA model and the "response" of the project stages with the four criteria.
- Develop a method that can be implemented in parallel with the proposed SQA model aimed at managing changes that might occur during the development process.

- Further examine the verification of safety-critical software and develop a set of protocols for the verification processes.

Finally, the proposed work in this thesis should be reviewed when a new version or notice of DO-178B is released or when a new standard is adopted. It should be also reviewed when new research results are published. It is recommended to implement the proposed work during the second phase of the CRIAQ project, which involves developments of further software components of the DTB. This implementation allows the investigation of possible changes in the lifecycle and in the SQA model.

Appendix A: Derivation of *P(S')* in terms of M$_t$

Since a failure in Safety-critical software may be tragic on human life, the defective rate *(θ)* should be extremely small. "Historically, this [extremely small defective rate] requirement has been translated into a probability of failure [*P(S')*] on the order of 10^{-7} to 10^{-9} for 1 to 10 hours missions" (Butler and Finelli, 1993), where a mission is the non-stop run time use of the software. This also suggests that *n* (the number of inputs) is better represented as a function of time, hence *n* can be expressed as:

$$n = ct$$

where *c* is the number of inputs per unit time *t*.

Moreover, since high accuracy is required in the case of Safety-critical software, *n* should be of a large magnitude.

When *n* is large ($n \to \infty$) and the failure rate is small ($\theta \to 0$) the binomial distribution can be accurately approximated by the Poisson distribution as follows (Walpole et al., 1998):

$$P(x) = \binom{n}{x}\theta^x (1-\theta)^{n-x} \approx \frac{e^{-n\theta}(n\theta)^x}{x!}$$

Therefore *P(S')* can be expressed as:

$$P(S') = P(x > 0) = 1 - P(x = 0) = 1 - e^{-n\theta}$$

The mean time to failure (M_t) is the average time that elapses before finding the first failure. From the Binomial Distribution properties, the mean number of failures (μ) is:

$$\mu = n\theta$$

Therefore, when *n* is a function of time as shown earlier the mean time to failure is:

$$M_t = \frac{1}{\mu}$$

Hence, when *P(S')* is computed as a function of time (*t*) (e.g. 1% chance of failure in 6 hours), it can be expressed as:

$$P(S') = 1 - e^{-\frac{t}{M_t}}$$

Appendix B: Requirements List Document

Project Name	
Component number	
Development Stage	

Label	**Name**	**Requirement Type**			**Description**	**Level**					**Is Complete?**	**Remarks**
		Syst.	High	Low		A	B	C	D	E		

Appendix C: Tasks List Document

Project Name	
Component number	
Development Stage	

Label	**Name**	**Level**				**Description**	**Is Complete?**	**Expected Output**
		4	3	2	1			

Appendix D: List of Documents

Project Name	
Component number	
Development Stage	

Label	**Name**	**Level**			**Description**	**Is Complete?**	**Is Clear?**
		3	2	1			

Appendix E: Functions List

Project Name	
Component number	
Development Stage	

Label	**Name**	**Description**	**Allowable margin of error**	**Output**	**Remarks**

Appendix F: Acceptance Document

Project Name					
Component number					
Development stage	**Completeness**	**Documenta tion**	**Intelligibility**		**Independence**
			Clearness	**Traceability**	
Project Initiation					
System Reqts					
SW Planning					
SW requirements					
SW Design					
Coding					
Code Testing					
Integration					
System Verific.					
Certif./Validation					

LIST OF REFERENCES

Airclaims Limited (2006) on behalf of British Civil Aviation Authority, "The World Aircraft Accident Summary database", Retrieved on December 22, 2005, from http://www.waasinfo.net

ANSI (2001), ANSI T1.523-2001.

ANSI/IEEE (1981), IEEE Standard for Software Quality Assurance Plans, ANSI/IEEE Std 730-1981.

ATIS (2000), Alliance for Telecommunications Industry Solutions, Telecom Glossary 2000.

Bersoff E. H., Henderson V. D., Siegel S. G. (1980), Software configuration management: an investment in product integrity, Prentice Hall, New Jersey.

Besterfield D. H. (2001), Quality Control (6th Ed), Prentice Hall, New Jersey.

Boehm B. W., Brown J. R., Kaspar H., Lipow M., MacLeod G. J., Merritt M. J. (1980), Characteristics of Software Quality, North-Holland, Amsterdam.

Boehm B. W. (1988), A spiral model for software development and enhancement, Computer, May 1988.

Bowen J. P., Vilkomir S. A., and Kapoor K. (2003), Tolerance of Control-Flow Testing Criteria, Annual International Computer Software and Applications Conference (COMPSAC 2003), Dallas, Texas, USA, November 2003, IEEE Computer Society Press, pp. 182-187.

Butler R. W., and Finelli G. B. (1993), Quantifying the reliability of life-critical real-time software, IEEE Trans. Softw. Eng 19, pp. 3-12

Cichocki T., and Górski J. (2000), Failure mode and effect analysis for safety critical systems with software components, Proceedings of the SAFECOMP 2000, 19th international conference on computer safety, reliability and security, pp. 382–394, Springer-Verlag, Berlin, Germany.

Crosby P. (1980), Quality is free, The Art of Making Quality Certain, Penguin Books, London, UK.

Deming W. E. (1986), Out of Crisis, MIT Press.

DO-178B (1992), RTCA Software Considerations in Airborne Systems and Equipment Certification, Washington, DC.

Dunn R. H. (1987), The quest for software quality, Handbook of software quality assurance, Van Nostrand Reinhold Company, New York, pp. 349-383.

Eckhardt Q. E. Jr., Lee, L. D., (1985), A theoretical basis for the analysis of multiversion software subject to coincident errors. IEEE Transactions on Software Engineering SE-11 (12): 1511–1517.

Ehrenberger W., and Saglietti F. (1993), Architecture and safety qualification of large software systems, Proceedings of the ESREL'93, European safety and reliability conference, pp. 985–999, Munich, Germany.

Elliott L., Mojdehrakhsh R., Tsai W.T., and Kirani S. (1994), Retrofitting software safety in an implantable medical device, IEEE Software 11 (1), pp 41–50.

Emmet L., and Bloomfield R. (1997), Viewpoints on Improving the Standards Making Process: Document Factory or Consensus Management?, Third International Software Engineering Standards Symposium (ISSES 97), Walnut Creek, California, USA, June 2–6, 1997.

Fagan M. (1976), Design and Code Inspections to reduce errors in software development, IBM Systems Journal 15(3).

Feigenbaum A. V. (1991), Total Quality Control (3rd Ed), McGraw-Hill, New York.

Fenton N. E., Neil M. (1998), A strategy for improving safety related software engineering standards, IEEE Transactions on Software Engineering, Vol. 24, Issue 11, November 1998, pp. 1002–1013.

Fisher M. J., and Cooper J. D. (1979), Software Quality Management, Petrocelli Books Inc., New Jersey.

Galin D. (2004), Software Quality Assurance, From Theory to Implementation, Pearson-Addison Wesley, New York.

Gardiner P. D (2005), Project management: a strategic planning approach, Palgrave Macmillan, New York.

Garvin D. A. (1987), Managing Quality: the strategic and Competitive Edge, Simon & Schuster, Canada.

Genuchten M. (1991), Why is Software Late? An Empirical Study of Reasons for Delay in Software Development, IEEE Transactions on Software Engineering, vol. 17, no. 6, June 1991, pp. 582-590.

Haney F. M. (1972), "Module Connection Analysis—a Tool for Scheduling Software Debugging", Proceedings of the AFIPS Joint Computer Conference, pp. 173–179.

Hooks I. F., and Farry K. A. (2001), Customer-Centered Products: Creating Successful Products Through Smart Requirements Management. Amacom.

IEEE (2002), IEEE Standard Glossary of Software Engineering Terminology.

IEEE/EIA 12207 (1998), Standard for Information Technology - Software Life Cycle Processes.

IPL (1997), Software Testing and Software Development Lifecycles, IPL Information Processing Ltd, Bath, UK.

ISO (1994), ISO 8402 : Quality vocabulary, Geneva, Switzerland.

Johnson J. (1995), Chaos: The Dollar Drain of IT Project Failures, Application Development Trends, vol. 2, no.1, pp. 41-47.

Juran J. (1951), Quality Control Handbook, McGraw-Hill, New York

Joseph J., Gryna F., and Bingham R. S. (1979), Quality Control Handbook (third edition), McGraw Hill, New York.

Kharchenko, V. S., Vilkomir S. A. (1999), Methodology of the Review of Software for Safety Important Systems. Safety and Reliability, The Tenth European Conference on Safety and Reliability, Munich-Garching, Germany, Vol. 1, pp. 593–596.

Kelvin W.T (1891-1894), Popular Lectures and Addresses

Kornecki A., Zalewski J., Ehrenberger W., Saglietti F., and Górski J. (2003a), Safety of Computer Control Systems: Challenges and Results in Software Development, Annual Reviews in Control, Vol. 27. No. 1, pp. 23-37.

Kornecki A., Zalewski J. (2003b), Design Tool Assessment for Safety-Critical Software Development, 28th NASA/IEEE Software Engineering Workshop, Greenbelt, MD.

Kornecki A., Zalewski J., Hall K., Hearn D., and Lau H. (2004a), Evaluation of Software Development Tools for High Assurance Safety Critical Systems, 8th IEEE Int'l Symposium on High Assurance Systems Engineering, Tampa, FL.

Kornecki A., and Zalewski J. (2004b), Criteria for Software Tools Evaluation in the Development of Safety-Critical Real-Time Systems, PSAM-7/ESREL'04 Int'l Conf. on Probabilistic Safety Assessment and Management, Berlin, Germany.

Kumar C. A. (1994), Excellence in Software Quality, India Infotech Standards, India.

Leveson N. G., Cha S. S., and Shimeall T. J. (1991), Safety verification of Ada programs using software fault trees. IEEE Software 8 (7), pp. 48–59.

Leveson N. G., Turner C. S. (1993), "An Investigation of the Therac-25 Accidents," IEEE Computer Applications in Power, July 1993, pp. 18-41.

Lewis, D. (1992), Computers and Translation. In: Christopher Butler (ed.) Computers and Written Texts. Blackwell, pp75-114.

Littlewood B., and Miller D. R. (1987), A conceptual model of multiversion software, The FTCS-17, international symposium on fault-tolerant computing, IEEE Computer Society Press, pp. 170–175.

Littlewood B. (1989), Predicting software reliability, Philosophical Transactions of the Royal Society, London, UK, pp. 513—526.

Maier T. (1995), FMEA and FTA to support safety design of embedded software in safety-critical systems, Proceedings of the ENCRESS conference on safety and reliability of software based systems, Bruges, Belgium.

Manns T., Coleman M. (1988), Software Quality Assurance, Macmillan Education, London, England.

McDermid J. A., and Pumfrey D. J. (2001), Software Safety: Why is there no Consensus? 19th International System Safety Conference, Huntsville, AL, USA.

Nagel P. M., and Skrivan J. A. (1982), Software reliability: repetitive run experimentation and modeling, NASA Contractor Rep. 165836.

NASA (1995), NASA Software Configuration Management Guidebook, Retrieved on November 3rd 2005 from, http://satc.gsfc.nasa.gov/GuideBooks/cmpub.html

O'Regan G. (2002), A Practical Approach to Software Quality, Springer-Verlag, New York.

Redmill F., Chudleigh M., and Catmur J. (1999), System safety: HAZOP and software HAZOP, John Wiley & Sons, New York.

Reifer D. J. (1985), State of the Art in Software Quality Management, Reifer Consultants, California.

Rubey, R. J., and Hartwick R. D. (1968), Quantitative Measurement of Program Quality, ACM National Conference 1968, pp. 671-677.

Schulmeyer G. (1999), Handbook of Software Quality Assurance (3rd Ed.), Prentice Hall, New York.

SEI (2000), The CMM Integration Model v1.02, CMM/SEI-2000TR018, Carnegie Mellon University, Pittsburgh.

Shewhart W. A. (1931), The Economic Control of Manufactured Products, Van Nostrand, New York.

Sommerville I., and Sawyer P. (1997). Requirements Engineering: A Good Practice Guide, John Wiley & Sons, New York.

Taguchi G. (1986), Introduction to Quality Engineering: Designing Quality into Products and Processes, Asian Productivity Organization.

Walpole R. E., Myers R. H., Myers S. L. (1998), Probability and Statistics for Engineers and Scientists (6th Edition), Prentice Hall, New Jersey.

Weinberg G. M. (1997), Quality Software Management Volume 4: Anticipating Change, Dorset House.

Wichmann B. (1999), Guidance for the Adoption of Tools for Use in Safety Related Software Development, Draft Report, British Computer Society, London, UK.

Wulf, W. A. (1973), Programming Methodology, Proceedings of a Symposium on the High Cost of Software 1973, Stanford Research Institute.