A Regression-based Model for Optimising Cost of Software Quality Assurance

Omar AlShathry

This thesis is submitted in partial fulfillment of the requirement

for the degree of Doctor of Philosophy

Software Technology Research Laboratory

Faculty of Technology

De Montfort University

July 2010

Declaration

I declare that the work presented in this thesis is original work undertaken by me for the degree of Doctor of Philosophy, at Software Technology Research Laboratory (STRL), De Montfort University, United Kingdom. No part of the material described in this thesis has been submitted for the award of any other degree or qualification. This thesis is written by me and produced using LaTeX.

Abstract

During the development of a new software system, project managers are always concerned about the triple constraints associated with the development process which are cost, schedule and quality. This ongoing concern emits from the fact that it is difficult to accurately quantify the trade-off process between these constraints. Software cost estimation models like COCOMO and COQUALMO and software quality process standards like ISO 9126 are used to predict software effort and defects estimation and to assess the quality of software being built. However, those models are based on data analysis of many previous software projects which may incur difficulties for an organisation to tailor any of those models to itself. Moreover, these models have not addressed the trade-off problem between the software triple constraints.

Cost of software quality (CoSQ) is a pressing concern for project managers as it has been estimated that around 40% of the software budget is spent unwisely on the defect detection and removal processes. The investment of quality improvements needs to be optimised in a way that does not affect the cost and schedule aspects. However, as is currently practiced in the industry, software artifacts, with respect to quality improvement activities, are considered equal in their significance and risk to the software development life cycle. The investment in activities concerning the detection and removal of defects is distributed evenly on the software artifacts without taken into consideration the risk and significance factors of such artifacts.

Our model gives the project manager the ability to control the investment given to the software QA plan by implementing optimisation techniques that are based on the data manipulation of historical projects. In addition, the project managers and QA practitioners relying on our model can handle and cope with unforeseen constraints related to their software development process. They can get optimal QA decisions to deal with budget shortage, schedule reduction or to achieve targets like a target of defect removal success, a minimal quality cost, etc.

Acknowledgements

First and foremost, I would like to present my deepest gratitudes to Almighty ALLAH for his bounties and blessings and for giving me the ability to finish this thesis.

I am indebted to my supervisor Dr. Helge Janicke who deserves special thanks for his support and guidance. Without his knowledge and valuable comments it would have been impossible to finish my thesis.

I would also like to express my gratitude to the director of Software Technology Research Laboratory (STRL), Prof. Hussein Zedan, for his unlimited support and advice throughout this work. I appreciate his positive comments for bringing about optimum consequences from my endeavors. I wish to thank Dr. Herb Krasner at the University of Texas at Austin for his insightful comments and advices during the initial year of my study.

I would like to thank the members of the STRL for the encouraging research environment and for the valuable suggestions and discussions.

My deepest gratitudes go to my beloved wife, I would like to express my loving appreciation to her for her tolerance and patience for being away from the family while I was studying. Many thanks for her encouragements and support during the PhD period without which it would not have been possible to finish this work.

I would like to extend my special gratitudes to my beloved parents for their invaluable constant support. I would like to thank Imam University in Riyadh for its financial support during my study.

List of Publications

1. AlShathry.O, Janick.H, Hussein.Z, Abdullah.A, *Quantitative Quality Assurance Approach*, In proc. of the IEEE International Conference on New Trends in Information and Service Science, NISS'09, 2009.

2. Abdullah.A, Janick.H, Hussein.Z, AlShathry.O, *Software Certification through Quality Profiling*, In proc. of the IEEE International Conference on New Trends in Information and Service Science, NISS'09, 2009.

3. AlShathry.O, Janick.H, *Optimizing Software Quality Assurance*, In proc. of the 4th IEEE International Workshop on Quality Oriented Reuse of Software (QUORS'10) in conjunction with the 34th Annual IEEE Computer Software and Applications Conference, COMPSAC '10, 2010.

4. AlShathry.O, *Software Quality Management System*, In Proc. of the 28th Annual Pacific Northwest Software Quality Conference, PNSQC'10, USA, 2010.

List of Acronyms

DRE	Defect Removal Efficiency
CoSQ	Cost of Software Quality
COCOMO	COnstructive COst MOdel
COQUALMO	COnstructive QUALity MOdel
ODC	Orthogonal Defect Classification
CMMI	Capability Maturity Model Integration
SPICE	Software Process Improvement and Capability dEtermination
MTTF	Mean Time To Failure
MTBF	Mean Time Between Failure
RCA	Root Cause Analysis
ROI	Return On Investment
SDLC	Software Development Life Cycle
QA	Quality Assurance
FP	Functional Points

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Objectives

- To identify the aims and objectives behind our research.
- To introduce the research questions and their motivations.
- To describe the research methodology used and adhered to.
- To highlight the contribution presented in this thesis.

1.1 Introduction

In the software development industry, the term "quality" is very common and considered to be the main factor that leads to satisfaction of the customers and hence to the success of the project. Software project managers believe that to get the competitive advantage the software should present high quality to the user. The importance of software quality is reflected as being an important factor of success to any software development project.

Another reason for this importance comes from the fact that the consequences of lack of quality can prove to be very expensive and severe.

In 1996 the world witnessed an unfortunate accident, which was the explosion of the Ariane 501 rocket just a few seconds after its launching from a French base located in the south of America. This unsuccessful launch had an estimated cost of more than $7 billion and several months of extensive work. A few weeks later after the incident, reports showed that the main reason which led to that catastrophe was an internal software error. The software controlling the navigation system of the rocket threw an exception due to unsuccessful data conversion from a 64-bit to a 16-bit format and this exception was not handled properly which caused a crash to the flight system. Further investigations showed that this software defect originated from a module which was re-used from previous release components for Ariane 400. The rationale behind re-using the module was that Ariane 400 had a successful launch and thus no testing was needed to prove the reliability and efficiency of the flight navigation module [35]. The project managers of Ariane 501 may have wanted to reduce the schedule of the project by re-using a vital module of a past release of the same project without exposing it to an extensive testing process to ensure it was bug-free.

This example shows how important quality is to any software development project. However, quality is expensive and results in the deduction of a considerable share of the the software development's budget and schedule. Budget and schedule are crucial aspects of any software development project and any negative impact on them may reduce the competitiveness required from improving quality.

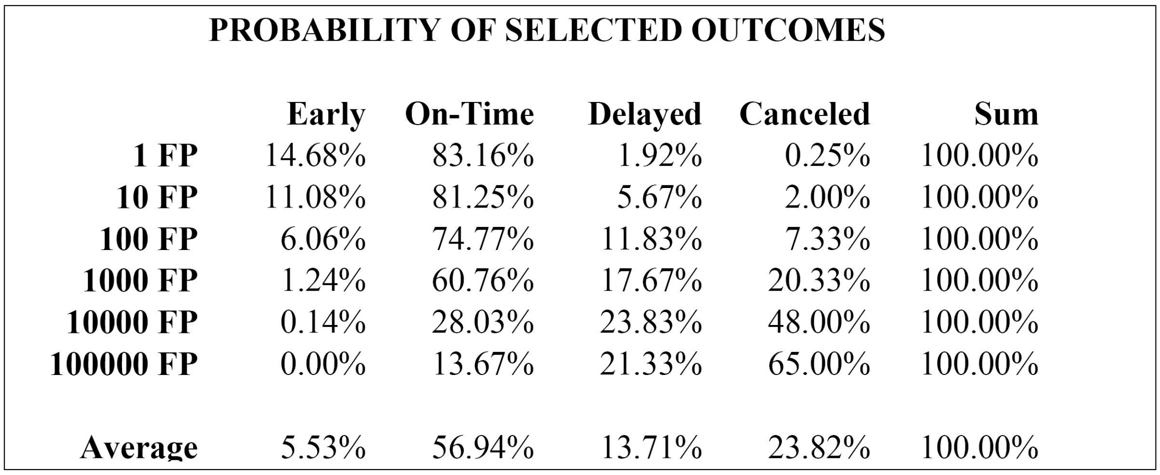
In any software development project, there are three triple constraints that need to be well-controlled and adhered to: cost, schedule and quality. A study suggested that 70%

of software project failures are due to three important causes: over-budgeting, unachieved deadlines and the sacrifice of quality [137, 76].

These constraints were not well-adjusted in the case of Ariane 501 where quality could have prevented that accident, but instead quality was sacrificed for a fast release and a reduction of cost.

Another important issue was that the sense of risk associated with such a software module of a high criticality was not considered well by the project managers of the Ariane 5. They should have taken into account the significance of that module to the whole project and subjected it to a substantial testing process.

Software development projects are one of the most costly and laborious projects in our time. A software project which exceeds 100,000 Functional Points (FP) which is equivalent to 15,000,000 lines of code, is estimated to cost hundreds of millions of dollars. Along with its high cost, software development projects have a high rate of failures compared with other projects (Figure 1.1).



PROBABILITY OF SELECTED OUTCOMES

	Early	**On-Time**	**Delayed**	**Canceled**	**Sum**
1 FP	14.68%	83.16%	1.92%	0.25%	100.00%
10 FP	11.08%	81.25%	5.67%	2.00%	100.00%
100 FP	6.06%	74.77%	11.83%	7.33%	100.00%
1000 FP	1.24%	60.76%	17.67%	20.33%	100.00%
10000 FP	0.14%	28.03%	23.83%	48.00%	100.00%
100000 FP	0.00%	13.67%	21.33%	65.00%	100.00%
Average	5.53%	56.94%	13.71%	23.82%	100.00%

Figure 1.1: Software Project Success Estimates: Source [78]

One of the main contributors to that high failure rate is the lack of effective Quality

Assurance (QA) practices to deal with software defects injected during the software development life cycle. It is estimated that most failed and canceled software projects are on time and work to a predefined scope until the system testing begins, when the project manager is overwhelmed by unexpected number of defects.

According to a recent study conducted by the US Department of Defense, the cost resulting from inadequate software quality to the economy of the United States is estimated at $59 billion per year [109]. This estimate shows the significance of software quality activities and the cost and consequences for the whole economy resulting from the inadequate quality of such software.

1.2 Problem

In order to be able to control the triple constraints of the software, the project manager needs to be able to know the consequences of each constraint over another. As our focus in this research is on software quality, the investment of quality improvements needs to be optimised in a way that does not affect the other two constraints. However, as is currently practiced in the industry, the software artifacts are considered equal in their significance and risk to the software life cycle with respect to quality. The investment in activities concerning the detection and removal of defects is distributed evenly throughout the software artifacts without taking into consideration the risk and significance factors. Some software modules hold risky and significant architectural components that need to be of a high quality and a low defect density. On the other hand, other modules do not require a similar level of quality. Defects originating from modules of high criticality may contribute to a project failure more so than less significant modules.

Current software quality tools like the defect containment matrix help the project manager determine the efficiency of QA processes applied in any software development phase by comparing defects found to defects escaped. However, this measure of efficiency overlooks the QA practices used during the QA process. Defects discovered are traced back to their sources only but are not linked to the QA practices responsible for their detection and removal. Therefore, the project manager relying on such a matrix cannot determine the efficiency of each QA practice in his/her organisation with respect to a specific development phase. Also, considering the fact that QA practices differ in their efficiency and applicability to a specific type of a software artifact, the project manager needs to know what the right QA practice to be applied is and at what time it is to be used. This will help in optimising the investment given to activities concerning the improvement of quality and will reduce the waste.

1.3 Aims and Objectives

The main aim of this research is to propose a model that helps the project managers to optimise the investment given to the QA activities of the software based on the triple constraints of the software and on the basis of the risk associated with the software development.

This aim leads to the following objectives:

1. To identify a categorisation scheme for software phases based on pre-defined risk levels.
2. To propose a data repository of QA practices to keep details of their defect detection and removal activities.

3. To propose an advanced defect containment matrix to help in identifying the defect removal efficiency of QA practices used.

4. To design an optimisation model using the Linear Programming technique that generates optimal solutions of QA plans based on given constraints.

5. To implement and evaluate the system using simulation of data from software projects.

1.4 Research Question

To build the direction of this research, a main research question was raised which was driven by issues in software quality and its associated cost. The main research question this thesis tries to answer is:

How can a software project manager have the ability to control and trade-off the triple constraints of the software: cost, time and quality, in a way that optimises the spending on QA activities ?

In light of this research question, more research questions were formulated in order to come to a full understanding of the current issues related to the main question and to help identify any difficulties that may have arisen during our research.

Such questions were :

1. How can a software development organisation build a knowledge base of their past QA activities to help them make informed estimates on future QA plans ?

2. How can investment given to QA activities be optimised to continue cost-effective QA activities ?

3. How to build a QA repository that contains QA data of historical software projects of the organisation ?

4. How to develop a system tool to support the management of QA practices ?

1.5 Original Contributions

This thesis makes the following original contributions:

- A Software Quality Management Model that supports the control of the software triple constraints.
- Updating and re-defining the Defect Removal Efficiency DRE) metric by considering the QA practices assigned within the QA process.
- A proposed adjustment to the defect containment matrix technique used by QA practitioners to monitor the QA activities within the software development life cycle that contributes to the body of knowledge of software quality.
- Formal expressions and statistical techniques that formalise the proposed Software Quality Management Model.
- Integrating an optimisation technique using the Simplex Method technique of Linear Programming to provide optimal solutions of QA plans based on pre-defined constraints given by the project manager.

1.6 Scope of Research

In this thesis, the main focus is to propose a system that helps the project manager and QA practitioners to make informative decisions on their QA processes in terms of their cost effectiveness and to trade-off alternatives of QA plans based on optimal solutions generated. A regression-based generic model is proposed that classifies each project's phase artifact to different work products according to pre-defined risk levels so as to prioritise investment given to the quality assurance process. The defect containment matrix which is currently used by QA practitioners is amended to include categorised work products and to take into account various defect detection techniques on the basis of their efficiency and cost.

It is worth mentioning that our proposed system which helps in quantifying the risk associated with software development projects mainly targets software projects of large sizes. Such software projects may require a software schedule of two or three years because of the extensive work needed. For that reason, it is difficult to evaluate our proposed Software Quality Management System using a real project due to the limited time dedicated to this research. An effort was made to collect QA data from projects that have already been developed in the industry. A contact was made with **NASA** and the **Promisdata** repository to obtain software data that allows the association of QA practices, their cost and efficiency during the different life cycle phases. However, they replied, stating that their stored data was not customised or classified according to our risk-based approach followed in this research. Instead, we used fictitious data simulating a real software project and we processed it using our system to show its functionality and its crucial role of making informed estimations regarding the investment of QA activities.

1.7 Thesis Structure

Our thesis is structured as follows:

Chapter 2 Presents a broad overview of quality perspectives and views considered by software engineers during the software development process. Highlights the quality process models like **ISO 9001** and **CMMI** and their roles in maintaining quality for the final product. An overview of the cost of software quality **CoSQ** and the models used to measure the cost associated with applying QA practices and the implications of the costs of defects.

Chapter 3 Describes the roles of QA activity in a traditional Software Development Life Cycle (SDLC) and the types of defect removal and detection techniques used. An introduction to the Defect Removal Efficiency (DRE) metric which is used to assess the success of defect removal processes and testing activities. A light is shed on the cost, quality and time relationship and give an introduction of the cost estimation models like **COCOMO II** and defect estimation models like **COQUALMO**, Capture-Recapture Models and Growth Models. Moreover, an overview is shown of the economic aspects of QA activities within the software development life cycle and what metrics can be used to work as incentives for QA plans.

Chapter 4 Presents our proposed Quality Management System Model and its architecture. It shows the different components of our models and the interrelationships between them and the risk-based software module categorisation process used. It introduces the structure of our system repository and how QA data is manipulated and stored to be utilised for future use.

Chapter 5 Introduces our mathematical model and its formal components. Statistical equations and formulas are proposed to fulfill each aspect of our cost variables, and to measure the effect of choosing a specific QA practice over another. A re-definition is made to the Return On Investment (ROI) metric and how it can be integrated within our model.

Chapter 6 Outlines the functionalities of our implemented tool and its components. It defines the instructions on how to use the tool in order to manage the QA activities during the software development process.

Chapter 7 In this chapter a hypothetical case study is chosen to simulate our system functionality. It shows how QA data can be represented in our system's repository and how to apply regression and optimisation techniques to produce informed estimates.

Chapter 8 Concludes and summarises the work proposed in this research, highlights the points of contribution that this research presents and discusses the limitations that were assumed. Moreover, it creates direction for future research in this field.

An overview of the dependencies between the chapters is depicted in Figure 1.2.

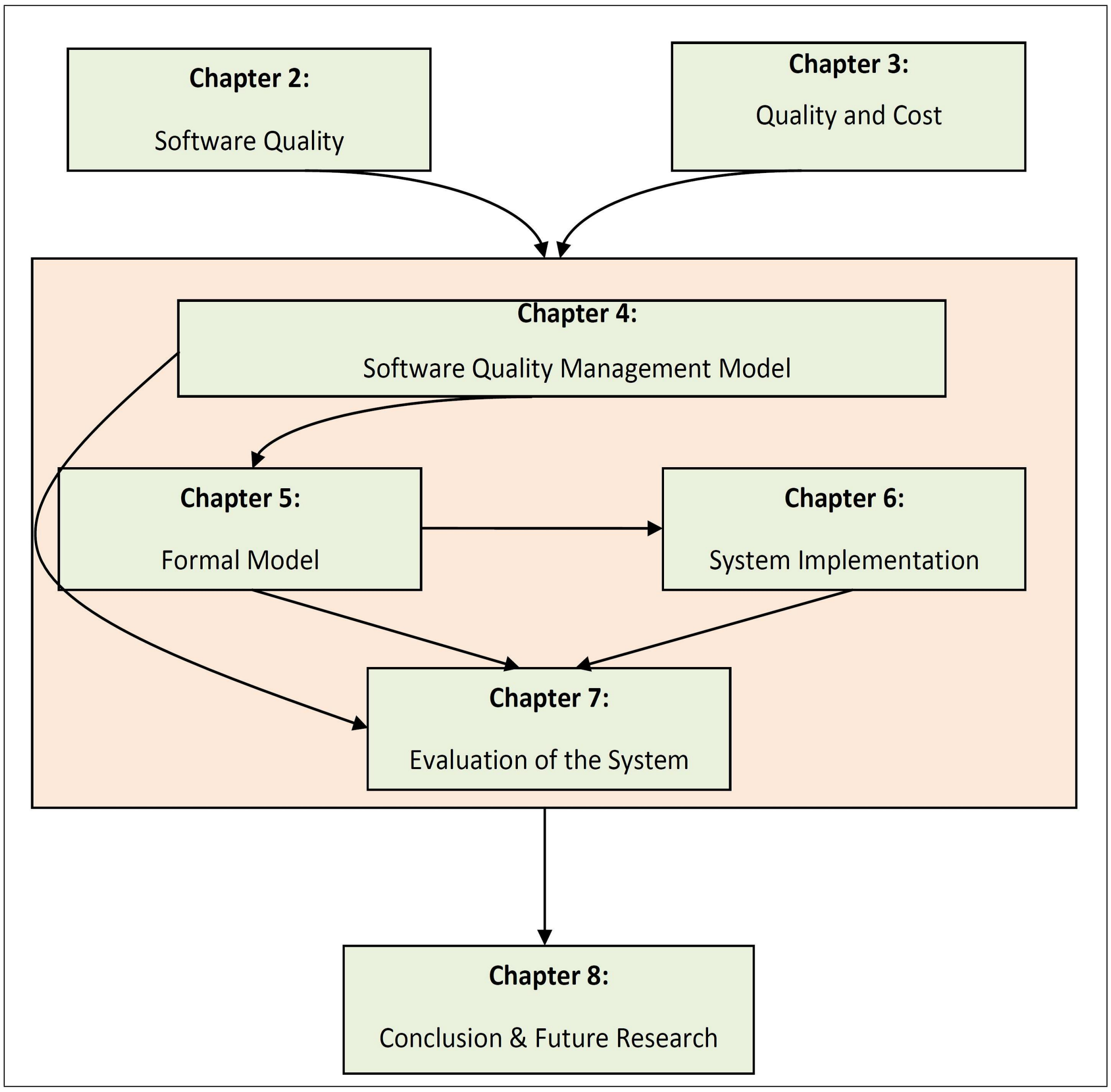


Figure 1.2: Chapter Structure

Chapter 2

Software Quality

Objectives

- To define the perspectives of quality and its views.
- To review the current models for software quality assessment.
- To discuss the defect detection and removal practices.

2.1 Quality

The term *quality* has many definitions and interpretations. For example, in the Oxford dictionary quality is defined as "the degree of excellence", while to the British Standard Institute *quality* means " the totality of features and characteristics of a product or service that bear on its ability to satisfy a given need." [21].

Also, quality is perceived differently from one person to another. For example, Crosby[31] defined quality as "the conformance to specification or requirements". That is, the software should perform what is expected of it with as little deviation as possible.

Juran[83] defined quality as the "fitness for use" focusing on the characteristic aspect of the product. Evans and Lindsey [43] interpreted quality as "the level of exceeding customer expectation". These expectations are usually implicitly defined by the customer. With respect to software, others define software quality as software which contains no defects or has a low defect density [107, 142, 33]. From the manufacturing angle, quality can be defined as the cost of removing defects [125].

Based on the different views discussed above, it would seem that there are two main categories of quality perception among people, these are characteristics and requirement (**Figure 2.1**). Requirements represent the needs that customers state and define, whereas the characteristics are those implied attributes the customer likes to get.

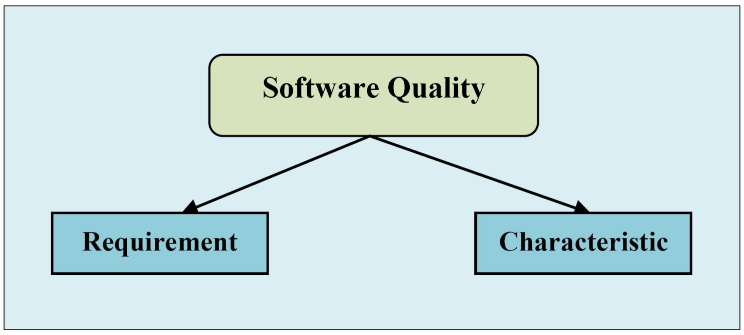


Figure 2.1: Software Quality Perception

For example, let us imagine that there is a software that totally conforms to its requirement specifications and has a very low defect density. However, the reliability of the software is not efficient and the software is not usable and the customers have some difficulties dealing with it. Based on Crosby's definition, this software is of good quality as it conforms to its specifications. However, according to the characteristics-based view of quality, this software failed to fulfill one major principle of quality, which is reliability, based on Garvin and McCall's models of quality [55, 101] as it may lead to the loss of customer satisfaction. So, when talking about software quality or quality in general, there must be a clear perception with regard to the two views of quality and to the links that lead each category to another. Also, when considering quality in software, the type of software needs to be taken into consideration and the circumstances related to its development.

There are other variations within the two views of quality in terms of the metrics and criteria used to assess the quality against one another. The following subsection outlines some of those different perspectives and how they differ from each other [77].

2.1.1 Quality Perspectives

In the software industry, quality is perceived differently from one person to another. This is related to the fact that there is no universal measurement for quality to be followed by everyone. In that respect, an overview is presented of the five aspects which mostly include all different perspectives of quality perceived by quality practitioners [55, 66, 125, 44].

1. Transcendental perspective

   Depending on this perspective, QA practitioners believe in the fact that quality is going to be acquired through experience and continuous works. There are no pre-determined guidelines or targets that the software should conform to.

2. User perspective

   In this perspective, the goal of quality is to meet the needs of users. Users can be those who are going to use the software after release or the stakeholders of the software projects. All users' preferences including the functionals and the non-functionals are taken into consideration during system development. Therefore, the extent of fulfiling those specifications according to this perspective is the quality criteria.

3. Manufacturing perspective

   Before the beginning of any new software development project, the project manager and the software developers meet with the stakeholders for the process of requirement gathering. The result of this meeting will be software requirement specifications which hold all system functionalities. The software under development must work according to these specifications as it would entail a reduction in development and maintenance costs. So, the goal of this perspective is the pursuit of software uniformity by conforming to requirements.

4. Product perspective

   With regard to the product perspective, quality of software is based on its internal properties and functionalities as well-built internal standards of the software leads to external quality.

5. Value-based perspective

   Software is not only how efficient and conforming to requirement it is; the relative cost and time needed to reach this goal is to be taken into account before system development begins. Increasing the quality of the software may not pay off enough after the software release. Mostly, this is the goal of software development projects, so investment made in improving software quality should be well analysed to calculate its revenue.

In accordance with the five views of quality mentioned above, views of quality have different influences on the software development process. This variation can be attributed to various reasons like the nature of the software development organisation, the project manager's point of view and also the proposed domain of the software. For example, military-based software will neglect the value-based perspective due to the fact that quality is crucial and missing quality may result in dangerous consequences. Therefore, when talking about quality, the type of product and the circumstances of the development process need to be taken into account.

2.1.2 Quality Assurance

Quality assurance (QA) is a term that is widely used along with software quality. According to IEEE, QA is "a planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements" [68]. This definition implies that quality is not the responsibility of the testing phase only. It is an accumulative process of practices in every phase of the software development life cycle. However, QA practices are diverse in their nature, applicable domain, efficiency and effectiveness which implies that the right choice of practice contributes to the desired efficiency of the QA process [152, 147]. In general there are two groups of QA practices: constructive and analytical:

- Constructive Quality Assurance

  In this type of QA, quality of software is related to its development process; therefore, constructive QA encompasses all activities, guidelines, tools and standards that ensure the quality of each phase's deliverables [37]. Examples of this type are model driven development approaches, code generators, Capability Maturity Model Integration (CMMI), Software Process Improvement and Capability dEtermination (SPICE) etc. Accordingly, it would seem that the sole purpose of constructive QA is to prevent defects from being injected early on in the software rather than detecting and removing them [9].

- Analytical Quality Assurance

  The main purposes of analytical QA practices are detecting and removing software defects rather than preventing them. Such analytical QA practices include verification & validation techniques, model checking and software testing [37, 107]. In other words, analytical QA is not a part of the software development process but rather its sole role is to verify each phase's artifact according to the software requirements specifications and prevent any deviation from them [142]. Despite the fact that a rigorous constructive QA plan can prevent defects, some defects may be overlooked and may propagate to the next stages where analytical QA plays its main role. Some analytical-based quality practices demand an executable code to perform their function. Conversely, some verification practices like formal inspections deal with most kinds of artifacts produced in the SDLC [149].

Generally, QA plays the important roles of assuring and controlling quality during the software development process. Such roles can be classified into the following:

1. Improving software quality by monitoring and auditing the software development process.

2. Assuring that software conforms and complies with the pre-defined standards and procedures and prevents any non-conformity.

3. Providing quick feedback to the software project managers once any deviations occur in the software development process.

2.2 Models of Software Quality

As showed earlier in Section 2.1, views of software quality can be classified into two aspects: product and characteristics. For each category, there are many quality models which have emerged in the past thirty years that establish the guidelines for achieving quality. This section outlines the common models of software quality and how these models can help in establishing software quality.

Generally software quality models are classified into two main families of models: process-based models and product-based models.

2.2.1 Software Quality Process Models

There have been some arguments concerning the absence of strong evidence that software process improvement models can increase the quality of the final software product [79, 121, 114]. However, many researchers, based on empirical studies, suggest that the quality of the application or the software is mainly related to its development process [38, 62, 7]. The more mature and effective the development process is, the lower the injected defects in the developed software, which in turn leads to an increase in the quality level and in budget saving [69, 7]. In the following text, the most common process improvement models that many software development organisations implement in order to improve the quality of their software are briefly described. Those models are: the SPICE model, ISO 9001 and lastly CMMI.

1. **SPICE Model**

   SPICE, which stands for Software Process Improvement and Capability dEtermination, is a process assessment framework developed by the Joint Technical Subcommittee between ISO (International Organisation for Standardisation) and IEC (International Electrotechnical Commission) [39, 61]. It was built to develop an international model for software process assessment. It has three main goals:

- To develop a draft for the standards of software process assessment.
- To conduct industry trials.
- To assert the efficiency of the process which leads to productivity and quality improvement. The SPICE model consists of six scales each of which is measured using process attributes that should be fulfilled using generic practices implemented in that process to move from one step to another Figure 2.2.

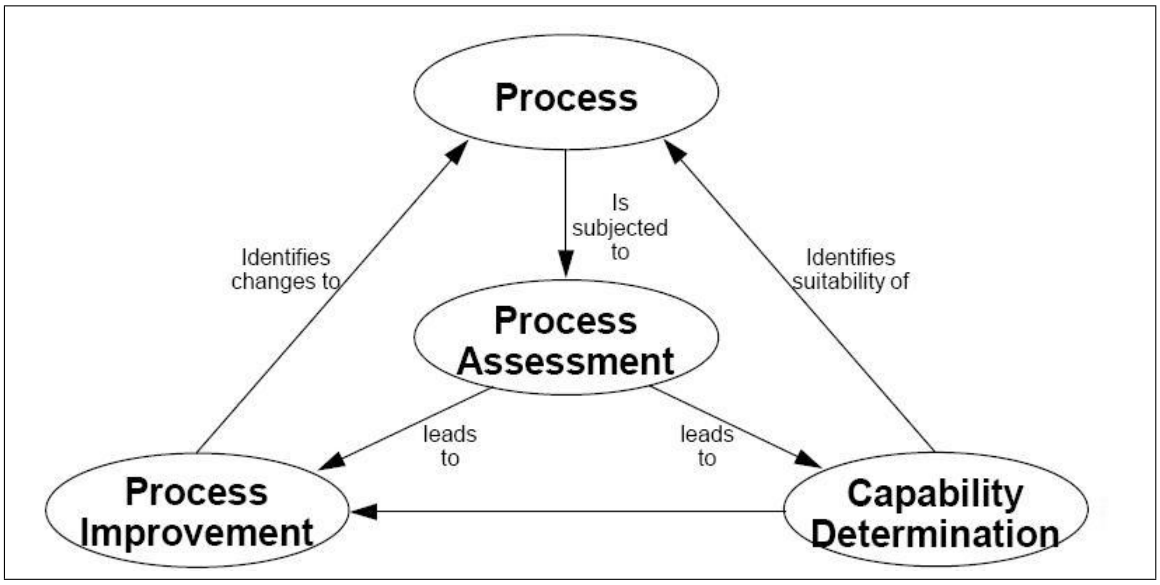


Figure 2.2: Spice Model :Source [135]

SPICE Levels are:

Level	Name
0	Initial. Not performed or incomplete process.
1	Performed. Informally performed process.
2	Managed. Planned process and traced process.
3	Defined. Well-established process.
4	Measured. Controlled and predictable process.
5	Optimised. Continuous improvement.

2. **ISO 9001**

   ISO 9001 is looked at as one of the most common quality establishments for process improvement [61]. It is currently used by more than 897,000 software organisations in 170 countries [71]. The sole goal of the ISO 9001 model is to integrate QA practices in design, development, production and installation. In other words, it is to enhance the continuous evaluation of the quality system for better achievement in the quality policy and its targets. It consists of generic standards and practices to assure QA in any organisation.

   An updated version of ISO 9001 is ISO 9001-3 which focuses mainly on how to apply ISO 9001 standards to software development [67]. It covers the following areas within the organisation:

   - Company and software management requirements.
   - Software project and maintenance requirements.
   - Supporting activities requirements.

3. **CMMI**

   CMMI or the Capability Maturity Model Integration is a process improvement approach developed mainly for software development projects [67, 29, 2]. It identifies the key practices and processes to improve the maturity of the software development process. CMMI is the successor of the Capability Maturity Model **CMM** which was developed by Software Engineering Institute SEI to help software organisation tackle the escalating problems of software quality and its relative cost [29].

Similar to the SPICE model, the CMMI model consists of five levels of a maturity process. Each level is characterised by the implementation of a variety of practices and techniques which help to achieve the desired development maturity of that level.

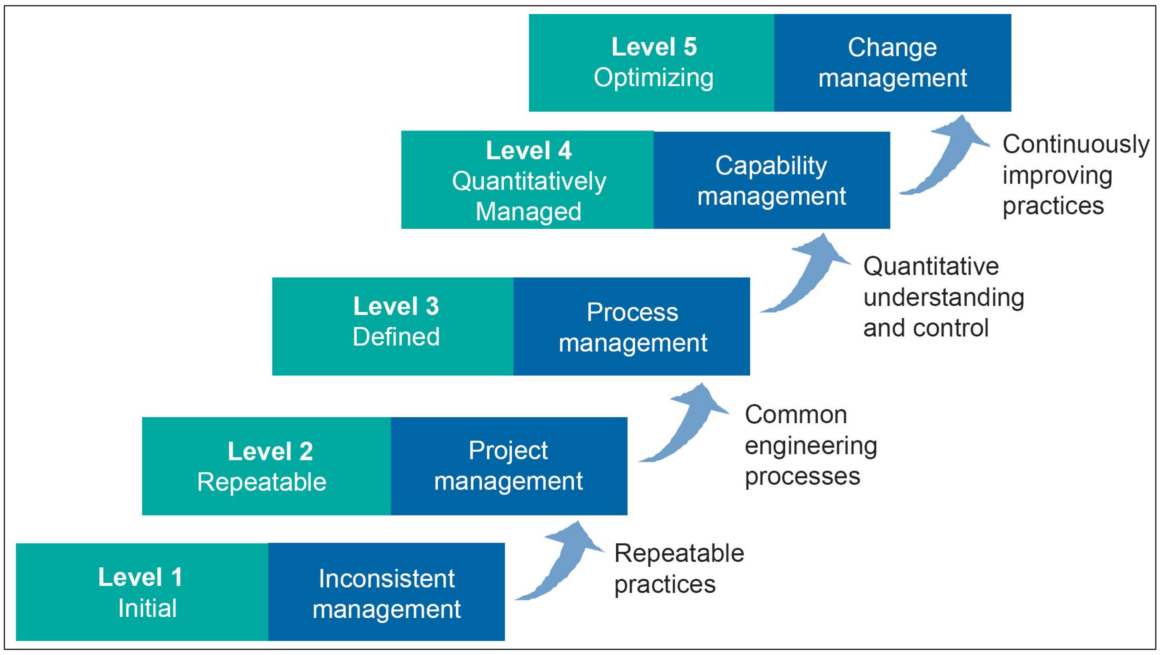


Figure 2.3: CMMI Framework : Source [32]

2.2.2 Difference Between CMMI and ISO 9001

Despite the fact that software process improvement models were introduced several years ago, no model has managed to replace the other nor has any organisation tried to implement the two models at the same time. This is related to the fact that there are some differences between those standards with respect to their applied practices, guidelines and maturity criteria. As an example, the CMMI model is targeted to system and software engineering, integrated products and process development [29], whereas ISO 9001 is more generic [61]. The concept of CMMI model is improving a product or an application through process improvement. ISO, on the other hand, is a quality management system focusing mainly on system requirements. There are also other variations like the implementation cost, which is less saved in the ISO 9001 [79], the assessment criteria, etc.

2.2.3 Software Quality Product Models

Software quality product models focus on the internal characteristics of the software under development [101, 61]. They describe the software as sets of components which have several properties through which the software is distinguished from other artifacts. Those properties are called software qualities or software attributes. Two of the most common quality product models will be expounded which are the McCall model and the Boehm model.

1. **McCall Model**

   The McCall model for software quality is one of the first software quality product models used by software development organisations and originated from the US military [101]. As shown in Figure 2.4 the McCall model consists of eleven quality factors on the left-hand side of the model; those factors describe the external characteristics of the software as perceived by its users. On the right-hand side, there are 25 quality metrics which are used to measure and quantify the 11 quality factors.

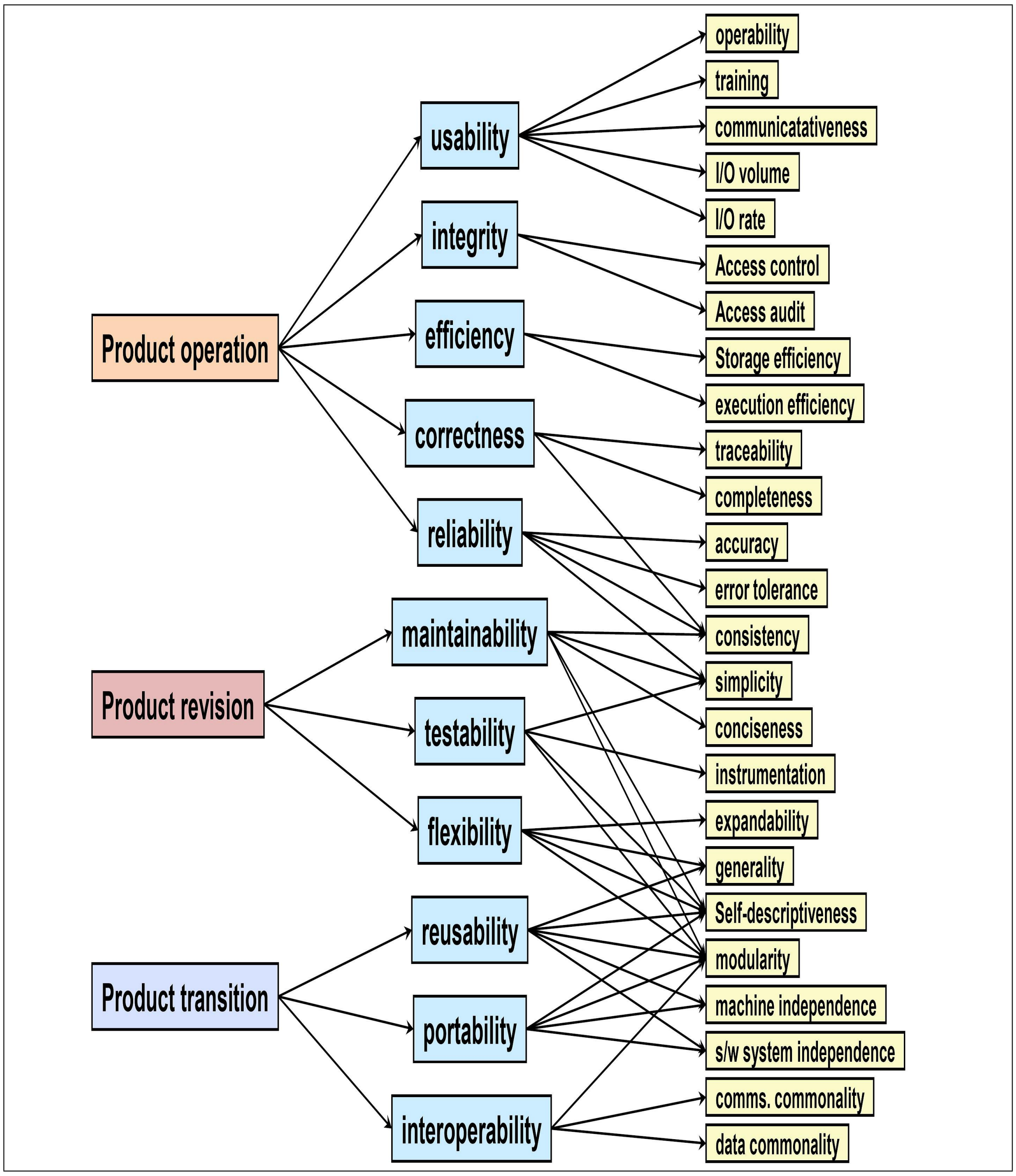


Figure 2.4: McCall Model for Software Quality Characteristics : Source [101]

As can be seen from the figure, each quality factor is measured by one or more of the quality metrics.

2. **Boehm model**

   Similar to the McCall model, the Boehm model defines quality as sets of attributes and metrics presented in a hierarchical way so that the quality of the software can be evaluated against them [61]. The only main difference between the McCall and Boehm models is that the latter model focuses mainly on the maintainability as a core quality factor for the software quality model [1].

Having discussed the well-known models of software quality, it can be noticed that such models encompass many characteristics and metrics that contribute to the final product, which in our case is the software. In this research, It is believed that estimating the accurate number of expected defects in a piece of software is a major challenge a software project manager faces. Therefore, the defect density metric is chosen as our perception of quality. The defect density metric is calculated as follows:

$$\text{Defect Density}= \frac{No.\ of\ defects\ in\ a\ code}{Size\ of\ the\ code}\ [84]$$

Referring back to the Boehm and McCall quality models discussed earlier, it can be seen that the defect density metric belongs to the reliability factor included in both models. Software reliability is defined as "the extent to which a program can be expected to perform its intended function with required precision" [53]. This metric is usually measured in the software industry by the defect density measure described above, as defects are the main contributor to a lack of precision and hence to the software's failure [105, 1].

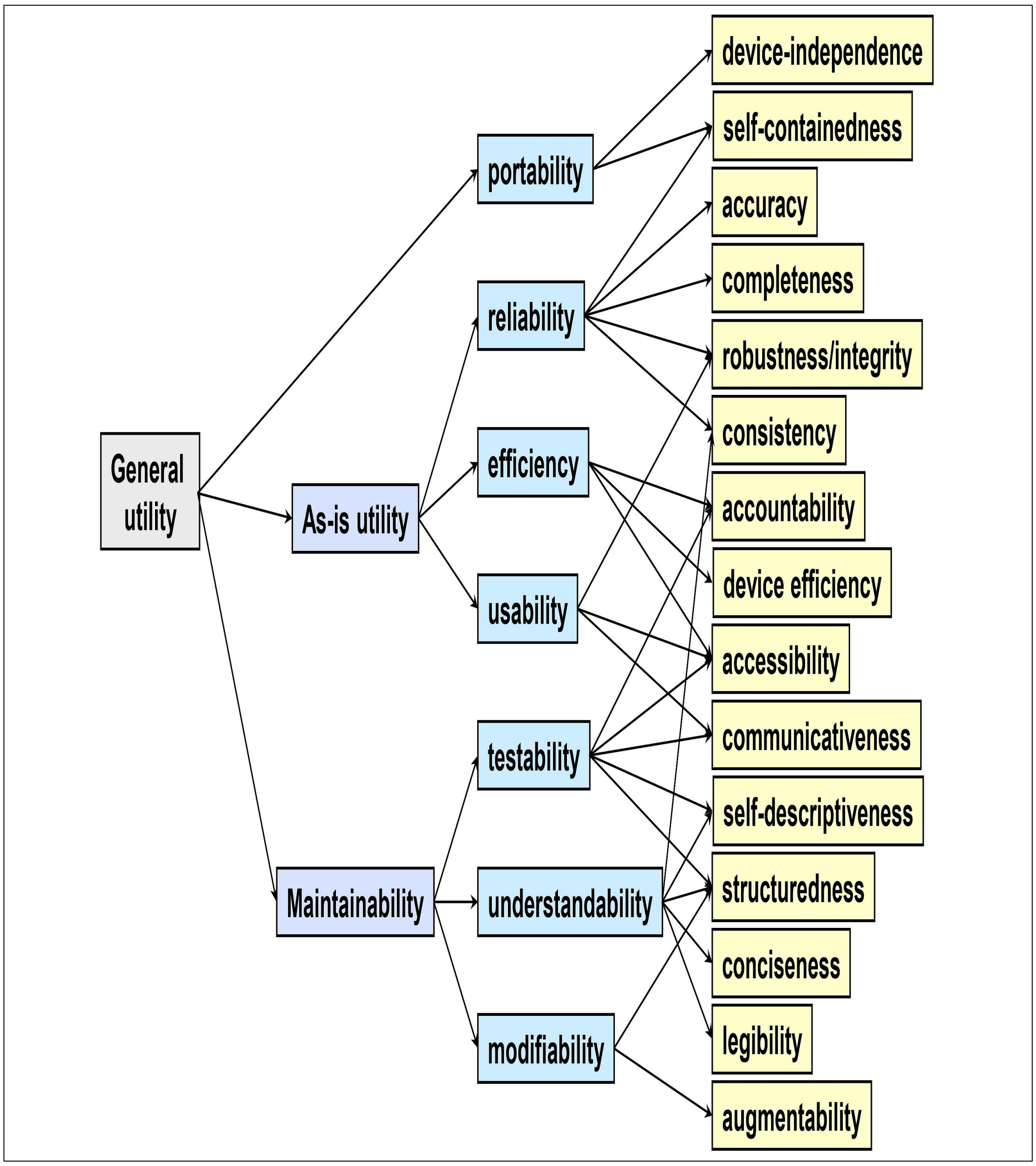


Figure 2.5: Boehm Quality Model : Source [61]

2.3 Software Development Life Cycle and QA

Software development organisations pay great attention to the QA activities during the software development process to help reduce the defect density of the final software. A high defect density value of a piece of software means that the number of defects injected are not proportional to the software size. Software defects have an enormous impact on software quality, software life cycle and customer satisfaction. This section gives a synopsis of the techniques and practices used for defect detection and removal activities and their allocation within the software development life cycle.

The system development life cycle, known as SDLC, is a methodology or a sequence of steps through which system developers build the software. In a nutshell, SDLC divides the process of system development into a number of phases or stages; each phase in turn is divided into further steps or sub-processes. There are different forms and versions of SDLC followed by software development organisations. The most traditional form of a software development processes is the Waterfall SDLC. In this methodology, the relationship between each phase and its predecessor resembles the flow of a waterfall [144, 34, 73]; that is, once a phase is completed, the software development process shifts to the other phase without returning to the previous one (Figure 2.6).

In any SDLC, there are four main phases which are: requirements gathering, design, code and system testing.

1. **Requirement**

   In the requirement stage, all system requirements in terms of the functional and non-functional are fully gathered and documented. This phase represents the backbone of the system as all of the following phases will be built upon the requirements

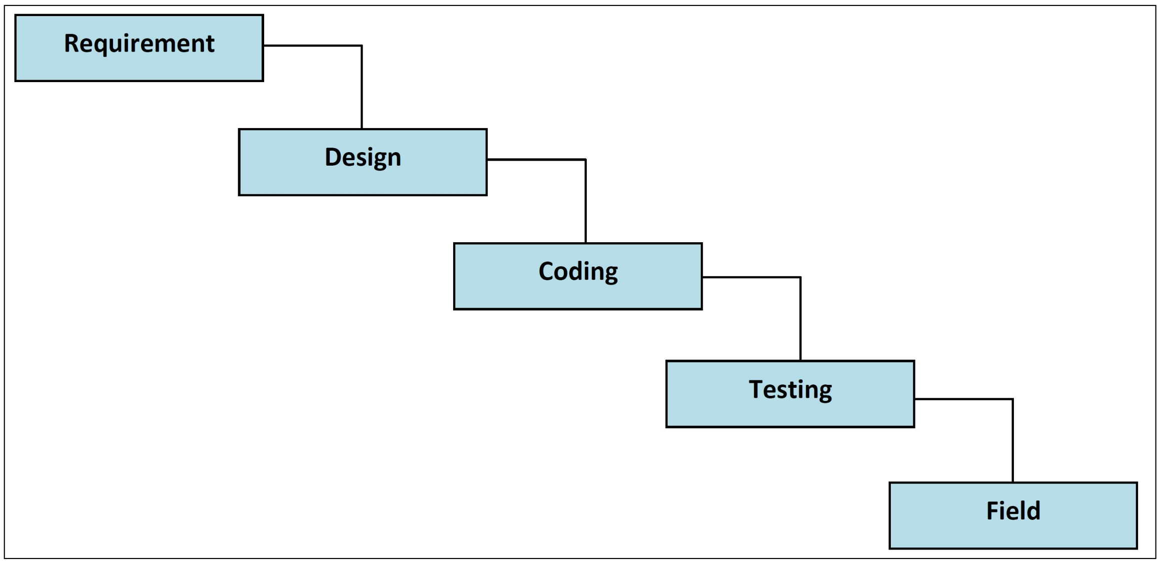


Figure 2.6: Waterfall SDLC

determined in that phase. The deliverables or artifacts of the requirement phase are called Software Requirements Specification (SRS) [144].

2. **Design**

   The design phase involves the architectural specifications of the system. Its main target is to move from the question *"what is?"* in the requirement phase to *"how to?"*. The deliverable of this phase is *Software Architecture Description* which describes the software in detail so as to make it easy for software coders to do their job. In some cases the design phase is divided into high-level design which includes use cases and design abstractions, and low-level design which includes artifacts that are ready to be converted to code, like class diagrams [34]

3. **Coding**

   In the coding phase the main implementation of the system occurs by converting the physical design taken from the design phase into a working application. The more accurate the design phase is, the minimal the additional input from the coders to perform the implementation or to make changes.

4. **System testing**

   The primary job of system testing is to uncover defects before the system release[104]. A defect can be any deviation from the main requirements the system is built for [14]. Many test cases are generated and run to ensure the integrity between system components and their modules. Once a defect is found, feedback is sent to the system developers in the coding phase in order to verify the defect and fix it. Sub-stages of the system testing phase and the variations among them are mentioned in Section 2.5.3.

Other versions of SDLC are: ***Spiral Model***, ***Rapid Prototyping*** and ***Extreme Programming*** [73, 34]. The main advantage of those versions over the Waterfall model is that they eliminate and reduce unnecessary documentation of the system [96]. Moreover, in many cases system requirements are difficult to collect and comprehend in the early stages of the system.

2.4 Software Defects

Similar to the variations of the definitions of software quality, software defects can be defined based on different views and perspectives [126, 48]. However, it is commonly agreed that a software defect is any flaw presenting in the software that prevents normal execution of the software, causing software failure or non-conformance to software specifications [14]. Usually, software quality practitioners employ common techniques like Root Cause Analysis (RCA) [29] and Orthogonal Defect Classification (ODC) [26] to determine the cause/origin of the software defects and to build a relationship between each defect and its source. However, it is argued that using one single technique to determine the origin of software defect is not sufficiently effective [17, 108] especially with the multiple variations in defects' attributes. The following subsection defines what these attributes are and how software engineers differentiate software defects from one another.

2.4.1 Defects Classification

Having agreed on the definition of the defect as being any flaw that prevents the expected outcome of the software, this definition does not imply that defects are identical or have the same impact on the system. Generally, based on a defect classification scheme of Orthogonal Defect Classification [115] and Fagan inspection techniques [20], software defects are classified according to four independent dimensions:

- **Defect trigger**
- **Defect source**
- **Defect type**
- **Defect severity**

1. **Defect trigger**

   Trigger is the technique that finds the defects or the environment the helps the defect to emerge. Such an environment would be the operational use of the software, another associated module or defects, etc.

2. **Defect source**

   With regard to the defect source, defect classification techniques try to trace the defect back to its phase source or in other words, which phase the defect originated from and when. In this case, the QA practice that should have found that defect and the specific module of the software this defect originated from, is overlooked which is an important issue our proposed research is trying to tackle.

3. **Defect severity**

   Defect severity is the potential risk a defect has if it is not fixed. In industry, software development organisations implement a top-down scale of defect severities in their QA activities. Comprehensively, Capres Jones [78] defined four main levels of severity of software defects which are:

   Critical: affects significant architectural components of the system preventing the functionality of the system.

   Major: affects system functionality and has a considerable impact on users.

   Minor: degrades system functionality and causes inconvenience to end users.

   Cosmetic: has no impact on system functionality and is more related to non-functional requirements.

   These different levels of defect severity support the notion of our research which is the need to optimise the resources given to quality improvement activities, especially when a shortcoming in the available budget and schedule occurs. This scale of severity levels is supported by another scale that classifies defects in terms of their type.

4. **Defect types**

   Defect types vary from significant functional component defects to simple defects related to non-functional aspects of the system. Below is a list of the major software defect types recognised by most defect classification approaches [26, 115]:

o FUNCTIONAL: Functional defects have an impact on software capability and require formal changes in the software architecture design.

o LOGIC: Impact on the module compatibility and functionality and require recoding of modules and components.

o INTERFACE: Related to integration and interaction with other system components.

o CHECKING: Affects program logic that would properly validate data and values before they are stored or used in computation.

o ASSIGNMENT: Minor errors in the program code that can be fixed by performing a few steps.

o TIMING/SERIALIZATION: Timing defects that can be resolved by adapting real-time resources.

o BUILD/PACKAGE/MERGE: Have an impact on the library system of the software and are fixed using software configuration techniques.

2.5 Software Defects and QA Practices

There is a great variety of QA detection and removal practices that are used and are available to software engineers. Those techniques differ in their efficiency and in their ability to find specific types of software defects [152, 147, 120, 124]. Before discussing these different types of QA practices and techniques, the responsibilities that QA practices have with regard to software defects will be briefly explained.

2.5.1 Responsibilities of Software QA Practices

Similar to the variations in defect types and their impact, QA practices that deal with software defects also differ in their roles and functions in the SDLC. Generally, the functions of QA practices and activities for software quality improvement can be classified into three main categories [54]:

1. Defect prevention

   Defect prevention deals with eliminating the potential of defect injection. In other words, it is a risk management procedure that reduces the possibility of introducing defects into the software [138, 120]. In defect prevention, many QA practices are implemented in order to prevent defects from being injected into the software. These practices are more focused on the analysis and design phases of the SDLC, i.e. before the coding phase. This role that QA practices play is crucial to the SDLC due to the fact that fixing a defect costs much more than the cost of preventing its occurrence, "*Prevention is better than cure*". Some examples of QA practices that serve this purpose are as follows:

   - Process improvement standards [104].
   - Formal methods such as formal inspection which is used to state the constraints and functions of the system's specifications and to verify the code to those functions [142].
   - Appropriate technical tools such as CMVC (Configuration Management Version Control) [104].

2. Defect detection and removal

   This includes all the QA activities that detect and remove software defects once they are discovered. Such activities are: formal inspection, testing and dynamic and static analysis [138, 54].

3. Defect containment

   In this approach some QA practices are implemented in order to contain failures at the end of the SDLC by limiting their damage. Defect containment techniques are mainly applied during the system release and at field stages whereas for defect detection and defect prevention techniques play their main role during the system release and field operation. Examples of defect containment practices are the fault tolerance techniques [142].

Despite the variations of the three categories of QA responsibilities, there may be some overlap between the first two roles of prevention and detection as there are some QA practices that prevent and detect software defects at the same time [104, 142].

As can be seen from Figure 2.7, which represents a typical waterfall development life cycle, defect prevention QA practices are linked to the first two phases of the SDLC which are requirements specification and design. On the other hand, defect detection activities are linked to the coding and testing phases as those activities, like testing, rely on executable code to perform their functions [104]. However, peer review, which is a typical QA practice for the requirements phase, is used to detect and remove defects rather than only to prevent them. Accordingly, it would seem that in each phase of the SDLC, there are QA techniques linked to defect prevention and others linked to defect detection. In this research, the emphasis is placed on the second role of QA practices, which consists of defects detection and removal activities.

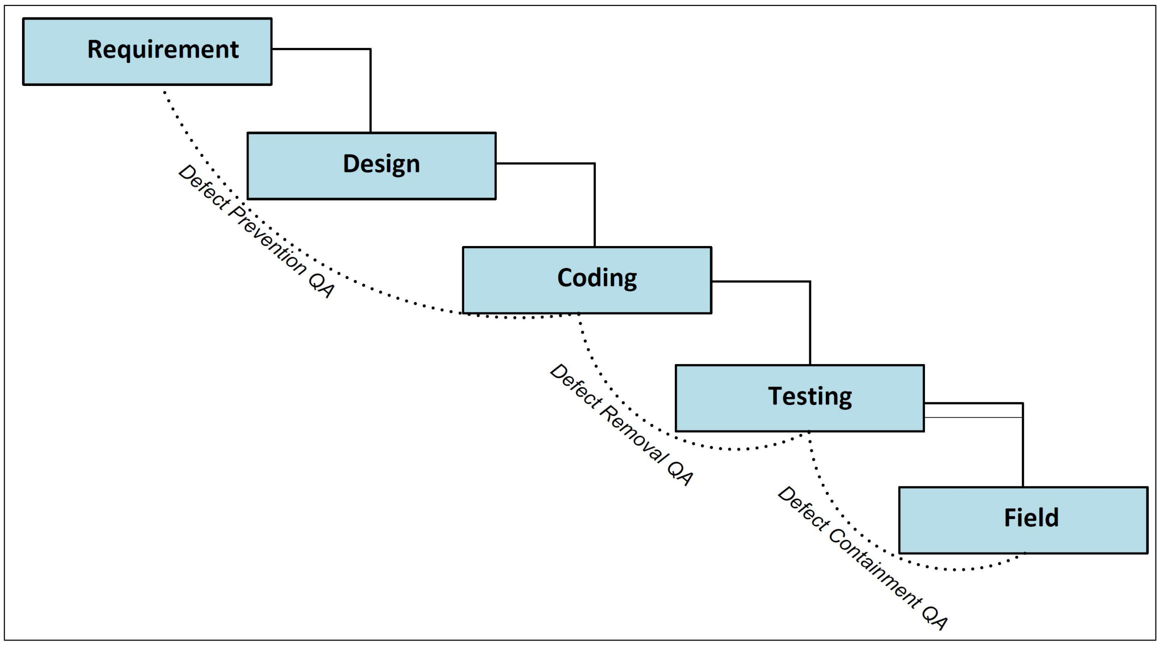


Figure 2.7: QA Activities Throughout SDLC

2.5.2 Types of QA Practices

This section outlines the types of QA practices that are used in the industry as some of them will be utilised in the evaluation chapter of this thesis. Mainly, there are two types of defect detection and removal techniques which are dynamic and static.

1. **Static Techniques**

   Static QA techniques are those techniques that do not demand any code for execution. Software QA practitioners can use them to check the conformance of the software to its requirements and for the defect detection activities.

   This important characteristic that gives the static techniques the advantage over the other QA techniques is that they are applicable to any phase artifact and deliverables of the SDLC phases: requirements, design and coding [120]. Along with the applicability feature of static techniques, their relative low execution effort and ease of use promotes their usage during multiple software development projects. Examples of such techniques are Formal inspection, Walkthrough and Peer review.

- Formal inspection

  Inspection is one of the most well-known QA practices that is used for defect detection and removal [54]. The distinctive feature that formal inspection has over any other QA technique is its proven efficiency in finding and detecting software defects in all types of software deliverables [124]; therefore, it is applicable to any phase of the software development life cycle [8]. Moreover, formal inspection is a well-structured process supported by well-validated frameworks like Fagan[112] and Gilb[112] inspection methods which makes it less erroneous and more accurate. In formal inspection 4 people usually participate in the inspection process, though it may take 8 people depending on the level of experience and expertise among participants. Generally, the responsibilities of formal inspection are as follows:

  (a) To discover errors in the function or logic implementation.

  (b) To ensure that the system works according to the functional requirements.

  (c) To store errors and defect data so that they can be avoided in future projects.

- Walkthrough

  Walkthrough is an informal process of inspecting some segments of software artifacts in a one meeting. The main difference between formal inspection and walkthrough is that walkthrough is led by a presenter whereas formal inspection is led by a leader. Moreover, formal inspection is a well-disciplined process consisting of pre-defined stages and phases, while walkthrough is more flexible with no formal structure or steps [54].

- Peer review

  Peer review, also known as technical review, is a less formal inspection technique where peers meet to review a work product in order to improve its quality. In peer review there is no attendance of managers and there are no pre-defined rules or steps to work to [54].

- Reading techniques

  In order to improve the accuracy of the QA process and to save the time and effort required, software QA practitioners usually use additional techniques and tools [90, 154] to be applied jointly with the QA practices discussed earlier. Such techniques are called reading techniques. Inspection reading techniques have evolved and developed through time and have proved their effectiveness in supporting the inspector in finding more defects in a shorter length of time [140]. Among reading techniques, there are some variations in terms of their efficiency and applicable domain [124, 110]. Moreover, some reading techniques might be suitable for some types of software artifacts and not suitable for others. Accordingly, researchers argue that the right selection of reading techniques considerably affects the efficiency of the inspection process [36]. Common inspection reading techniques are:

- Ad-hoc reading
- Checklist reading
- Stepwise abstraction

- Scenario-based reading
- Perspective-based reading

In the following, some of the reading techniques that are currently used are defined for the sake of comparing them and showing the differences between them as they will be utilised later in the evaluation chapter of our model.

(a) **Ad-hoc reading Techniques**

The ad-hoc reading technique depends entirely on the inspector's skills and knowledge. In other words, it gives little or no support to inspectors as there are neither general rules to be followed nor an inspection procedure throughout [27, 91].

(b) **Checklist Techniques**

In the checklist reading technique, the inspector has a list of issues to check against during the inspection process. In this case the inspection process is well-supported compared to the ad-hoc techniques by having pre-defined rules and responsibilities which help in centering the attention in the inspection process on certain issues which, in turn, help to find defects easier [90, 110].

(c) **Scenario-based Reading**

This approach is built upon using a scenario which is either question-based or description-based to help the inspector to focus on a specific defect domain. In this process, defects are classified into groups of certain types and the team leader develops a scenario for each defect classification separately. As each team member of the inspection process would follow a separate scenario that looks for different defects, it is argued that the efficiency of inspection would be better compared with other reading techniques [154].

There are three types of scenario based reading techniques:

(a) **Usage-based Reading**

Developed by Olofsson and Berge [139, 141], the usage-based reading techniques alert inspectors to important parts of the software from a user's point of view and scales back the impact of defects found in the system under development.

(b) **Perspective-based Reading**

A common type of scenario-based technique where software artifacts are produced according to three perspectives: tester, designer and user [154].

(c) **Defect-based Reading**

In a defect-based scenario, the reviewing process is created in order to detect a specific class of defects [141, 124].

2. **Dynamic techniques**

Dynamic techniques, known as testing techniques, are techniques which rely on the execution of code to uncover defects [6, 120]. This definition gives the implication that the main usage of dynamic techniques starts at the testing phase of the SDLC. There are many types of software testing techniques which differ in their efficiency and applicability. A remarkable variation between software techniques is the level of opacity [104], which is the way in which the tester using that technique looks at the code.

There are two general opacity levels in software testing: either black-box or white-box.

(a) Black-box testing

Black-box testing, also known as functional testing, treats and views the system as a black box without looking at its internal structure [120]. It is limited to viewing and comparing the input and output of the software to verify that the system requirements are being met. Types of black-box testing are:

- Equivalence Classes and Input Partition Testing
- Boundary Value Analysis

(b) White-box testing

Unlike black-box testing, white-box testing, or structural testing, looks into the internal control of the software and its sub-modules to generate test cases based on their interrelationships [6, 104]. White-box testing gives the tester the level of complicity to deeply analyse the software control flow and objects of applications [120]. Types of white-box testing are:

- Control flow testing
- Data flow testing

2.5.3 System Testing Phase

Despite the variety of QA practices and their comprehensiveness in all stages of the software life cycle, the system testing phase remains the main defect detection and removal stage in any software development project [6] (Section 2.3). In this phase, testers perform repetitive testing activities in order to minimise the injection of defects into the software before its shipment [120]. There are many testing activities with different levels and depths which occur in this phase [11]. In general, the most common testing activities, which include either black-box techniques, white-box techniques or both , are:

1. Unit testing

   Unit testing is the analysis of individual units or functions of the software for the purpose of uncovering defects and errors [120, 6].

2. Integration testing

   In integration testing, the testers group compose the software modules in order to verify the interaction between them [120].

3. Regression testing

   Regression testing is a re-testing process where some pre-tested modules of the software which were modified are re-tested again to ensure their compatibility and conformance to the software requirement [120, 104].

2.6 Efficiency Problem of QA Practices

Despite the diversity of test methods and techniques available in the software literature, there are some ongoing problems that hinder software quality engineers from achieving their quality target of finding most of software defects. One of the main related problems is the right selection and suitability of a QA practice to a specific module of the software [149, 82]. A main driver for this problem is the notable variations in the efficiency of QA practices and their relative cost level and execution time [84]. Another problem is that there is profound confusion at the final stages of the software development process as to which defect discovered in the system testing phase belongs to which development phase of the software's SDLC [78] (Figure 2.8). Techniques like the Root Cause Analysis (RCA) [29, 63] and the Orthogonal Defect Classification (ODC) [26] can be applied jointly during the software development process to determine each defect's attributes in terms of its type, trigger, source, etc.

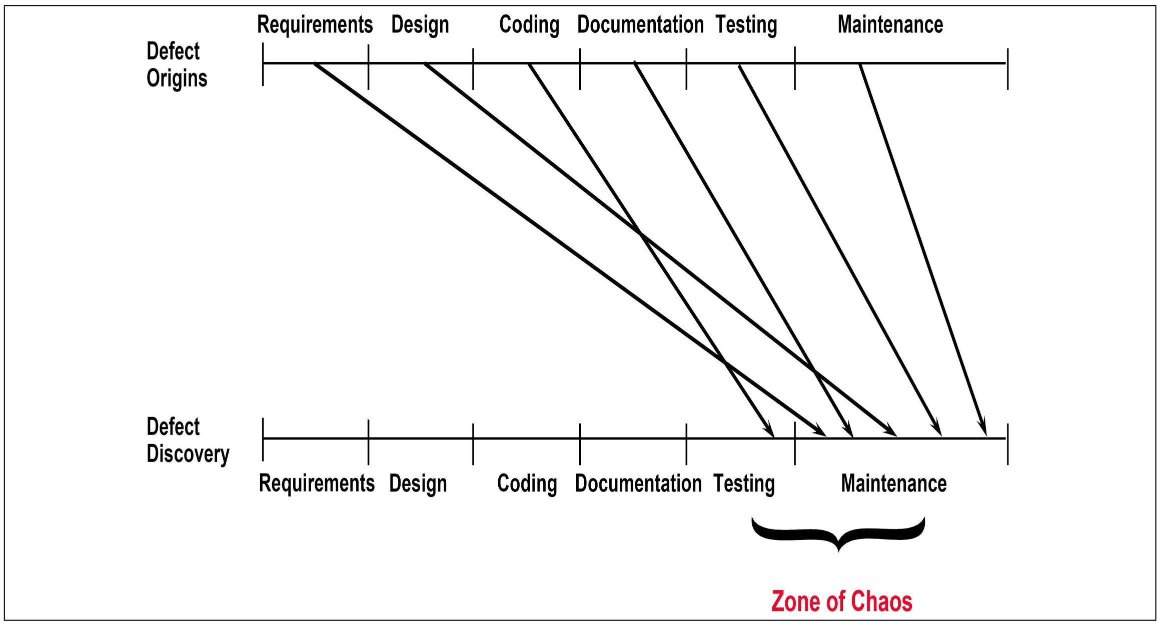


Figure 2.8: The Defect Origin Tracing Problem : Source [78]

However, in order to plan efficient defect detection and removal activities, there is a need to find a mechanism to determine the origin of any detected defect - not only the part of software it originated from, but also in terms of the verification technique applied. That is, at the development phase of the software development life cycle, some defects will manage to escape from the QA practices applied in that phase to the later phases. Once these defects are discovered at the system testing phase, they should be sourced back to the technique that was responsible for their detection and removal in order to determine its efficiency. Doing so will determine how efficient and suitable a technique is to that particular phase and what effort it needed to achieve its task.

In this research, as will be shown in Chapter 4, a model is proposed that helps to overcome the traceability issue in a way that any defect found is directed and linked not only to the technique that it escaped from but also to the part of the software it belongs to. This mechanism will help QA practitioners determine the efficiency of QA in dealing with a specific part of the software (Figure 2.9).

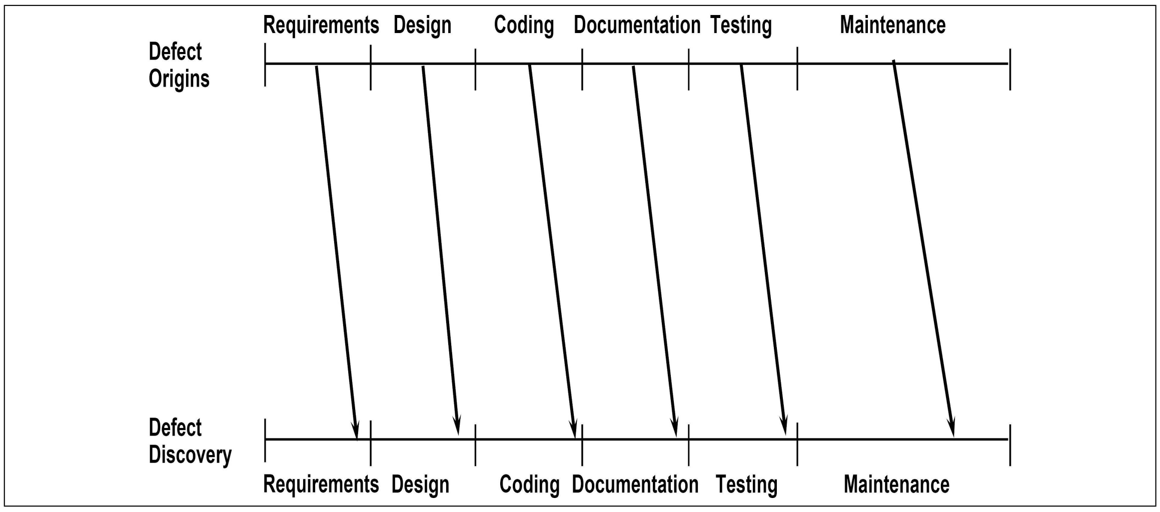


Figure 2.9: Defects Traced Back to Their Origins

2.6.1 Defect Removal Efficiency

As earlier discussed in Sections 2.5.3 and 2.5.2, there are plenty of defect detection and removal techniques: dynamic and static, white-box and black-box, scenario-based or ad-hoc based, etc. With all of these different types of defect detection techniques, project managers and QA practitioners need to continually make decisions on which technique is suitable to be applied and when [30, 82]. Usually at the end of any software development process, questions are raised by the project manager regarding the success of the QA activities conducted during the software development process. Such questions center in the context of how many defects were removed and how many defects were passed on to the end user. In order to answer such questions, QA practitioners implement some metrics that help to measure the success/failure rate of defect detection and removal processes. The most common metric is called software defect removal efficiency, which is referred to as DRE.

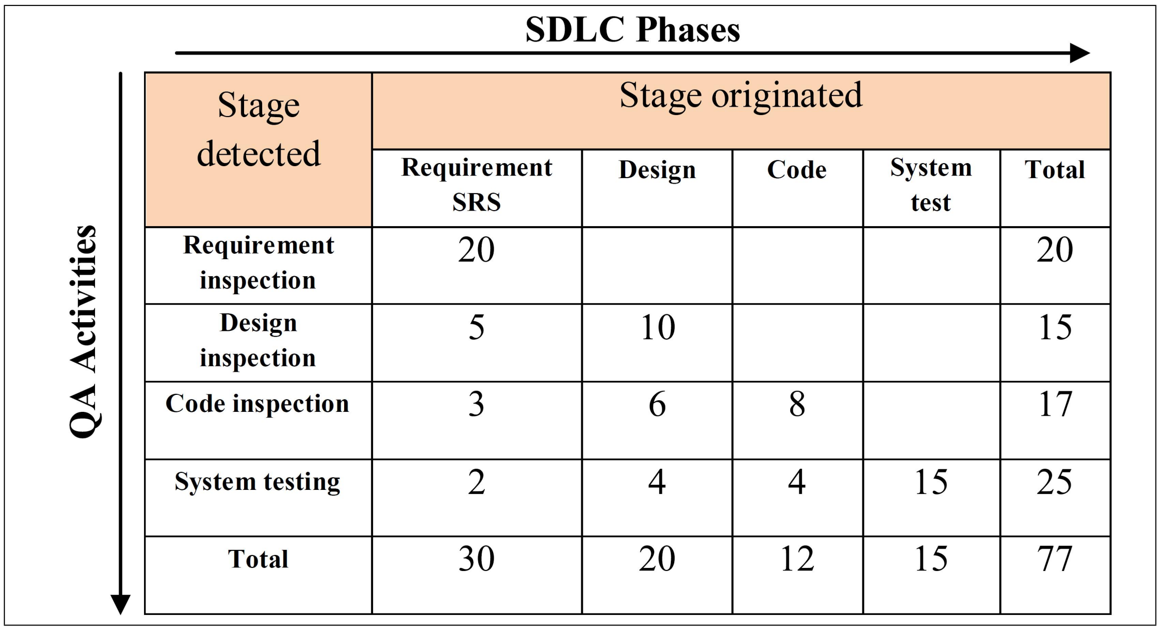
DRE is a software QA metric which measures the defect removal effectiveness of any software development phase by comparing the number of defects found and removed before release to the number of defects found after release [30]. The DRE is calculated as a percentage value (%), so that the higher the value is, the more effective the defect detection and removal for a specific phase or for the whole development life cycle. Accordingly, this leads to an optimistic view that the project will carry on without quality and schedule problems [25, 132].

2.6.2 Defect Containment Matrix

To accurately calculate the *DRE* for each phase in particular and for the whole SDLC, companies apply what is called a defect containment matrix in order to analyse the efficiency of their defect verification & validation techniques [84, 30]. The matrix (Figure 2.10) was applied to track the defect injection and removal activities; that is, in which phase a defect was inserted and in which phase it was removed. Applying this technique for the whole SDLC, the software project manager will be able to analyse the efficiency of defect removal activities as a whole for each software development step.

DRE = Found/Created = X%

As can be seen from Figure 2.10, the defect containment matrix consists of one row which represents the normal software development stages, and the first column represents the QA practices applied at each stage. Numbers inside the inner boxes indicate the number of defects found by each QA practice with respect to the stage it is being applied to. Some defects are found and corrected at the same stage (10 defects in the design phase), while some defects found after the stage is completed (6 defects found by code


SDLC Phases
QA Activities

Stage detected	Stage originated				
	Requirement SRS	Design	Code	System test	Total
Requirement inspection	20				20
Design inspection	5	10			15
Code inspection	3	6	8		17
System testing	2	4	4	15	25
Total	30	20	12	15	77

Figure 2.10: Defect Containment Matrix (Example)

inspection) also belong to the design phase. In order to measure the success for any QA practice, the DRE metric is used. As an example, the DRE for applying requirement inspection based on (Figure 2.10) equals:

$$DRE = \frac{20}{30} \approx 66.7\%,$$

This value means that around *66.7%* of the defects created in the requirements stage were successfully removed by the inspection practice. Based on that, the project manager would use this value as a baseline or a standard to not only evaluate the current process but also to evaluate future projects. In the future, for any similar software projects or software post releases inspected with the same inspection process and having all relevant factors like people, tools and techniques used in a stable manner, the resulting DRE value will be compared with the previous value of 66.7% as a baseline to assess its success [30, 84]. Another advantage of the defect containment matrix is that a low DRE value of a specific QA practice will help the project manager implement new improvements to the

QA process early in the software development process. These improvements would be reflected in considerable savings in the project budget and schedule.

Despite the fact that the defect containment matrix has a wide acceptance among QA engineers, applying and building QA decisions entirely on such a matrix lacks the required accuracy and effectiveness. The reason for that is the efficiency value given by the DRE is general for the entire defect removal process without considering the techniques used specifically [4, 5]. Therefore, project managers will not be able to compare the efficiency of each verification technique applied because of the inability to measure individual techniques for each phase. Another ongoing problem of the current defect containment matrix is that measuring the success/failure of a QA practice in removing defects using a percentage value is biased and not convincing due to the differences in defect types and their severities [4]. The latter concern applies to most process control practices that rely on a DRE value as a success criteria of QA practices. As discussed in Section 2.4, software defects vary in their severity in terms of the magnitude of the negative impact on the testing process and on the whole system [26, 84].

There are major defects that affect significant architectural components of the system and there are cosmetics which are related to non-functional attributes of the software (Figure 2.11). In other words, defects originating from critical and important parts of the artifact cannot be compared with trivial defects, as those important or critical defects take more iteration and time to be corrected and re-worked.

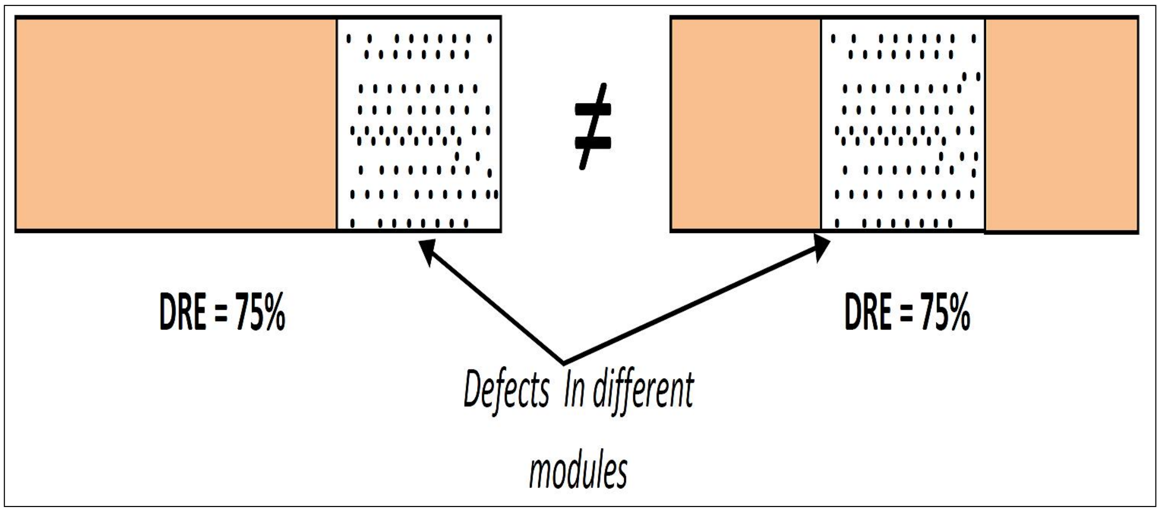


Figure 2.11: Different Considerations of DRE Values

2.7 Summary

This chapter reviewed the different software quality views perceived by QA practitioners, It gave an overview of the most common models of software quality, the software process improvement models and how process-based models differ from the product-based models of software quality. A short overview was given to the roles and responsibilities of QA in the SDLC and how it can mitigate the effects of the injection of defects into the system testing phases. This chapter also discussed the criteria used for evaluating the success of QA processes and highlighted the problems currently associated with it. A detailed critique of the defect containment matrix technique was provided which emphasised the shortcomings of such a technique. Generally there are two main shortcomings that were found upon our revision of the earlier sections:

- The DRE value is generalised and does not account for the variation of defect types and severities.

- The traceability problem of the efficiency of a QA practice with respect to module of the software it is applicable to.

In the following chapter, an outline of the relative concepts related to our research questions is identified. A definition to the triple constraints of software development and their roles during any software QA process is given. Also, a literature review is given to the related models that are used to link the triple constraints and hence optimise investment in software quality and their limitations.

Chapter 3

Quality and Cost

Objectives

- To review the CoSQ principle and its association with QA practices.
- To highlight the relationship between the software triple constraints.
- To discuss state of the art literature related to our research questions.
- To highlight the contribution presented in this thesis.

3.1 Introduction

The previous chapter defined software quality and outlined its importance to software development organisations as a main factor of success in software projects. However, the importance of software quality should not result in overlooking the high cost it introduces. Therefore, decisions regarding increasing and decreasing quality are to be considered carefully as quality is not free and increasing quality entails an increase in the cost and timescale. During software development projects, a debate usually occurs between software quality practitioners and software project managers on when is the most suitable time to stop testing and release the product to end users. The factor causing such debate is the difficulty of quantifying the cost effectiveness of increasing the investment in implementing QA practices and defect detection techniques during the SDLC stages [97, 92].

Software development organisations have substantial interest in having preliminary details and estimates of the cost that would be introduced by implementing quality improvement initiatives for their software [131, 88, 130]. These estimates are of high importance to the software project managers in order to compare them against the overall development cost of the software and hence to determine the feasibility of any future quality improvement plans.

The following sections will briefly explain some of the models pertaining to the cost of software quality analysis and the relationship that links the cost of software quality with the overall quality and schedule dimensions.

3.2 Cost of Software Quality (CoSQ)

The term CoSQ, which refers to the cost of software quality, describes the trade-off mechanism between delivering software of a high or an acceptable level of quality and the cost associated with it [88, 92]. It is a mechanism that helps software project managers determine the accepted and affordable level of quality for their products. Moreover, CoSQ helps QA practitioners find an answer to the inevitable question raised by software project stakeholders about the cost savings that would be gained that would offset any expenditures on quality improvement [133]. Also, it helps software project managers to determine the possible alternatives of quality improvements practices [64] and to make a comparison between them on a well-defined basis. Generally, there is no validated and agreed-upon cost of software quality model today. Most software organisations use a quality cost model taken from manufacturing and adapted to be applicable to software. This model classifies the cost of quality into the cost of achieving quality and the cost of poor quality [150, 143, 88, 92].

3.2.1 CoSQ models

1. **PAF model**

   The most well-known cost quality model in the literature is what is called the *PAF* model [122, 88, 131, 72]. The *PAF* model was originally used in manufacturing industry and then was adapted by software engineers to be applicable to measure the cost of quality activities in software development.

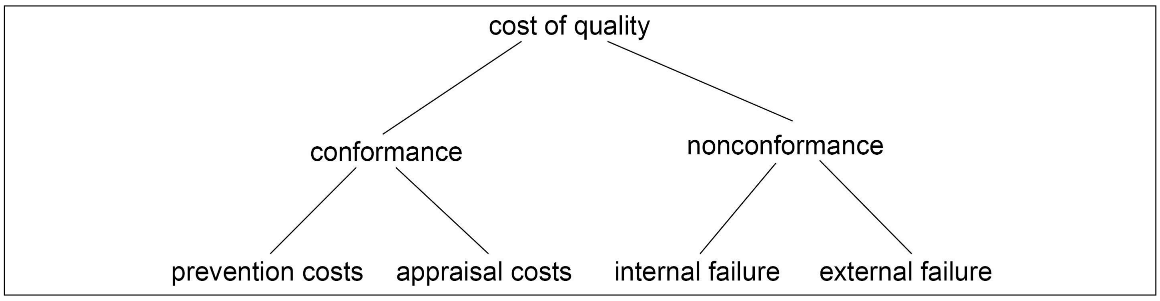


Figure 3.1: PAF Model Philosophy

   According to the *PAF* model, as seen in Figure 3.1, the cost of quality is measured as the sum of conformance and non-conformance costs. Conformance cost, which is divided into prevention and appraisal costs, is the cost related to implementing practices and techniques to prevent poor quality. For example, implementing process improvement models like CMMI [37] or using code generator tools to help minimise the introduction of errors are two kinds of prevention costs. On the other hand, the cost of applying formal inspection for software project deliverables or the cost of document reviews are considered to be appraisal costs as they prevent defects from propagating to later phases. Non-conformance cost is the cost of quality failure or the cost of not conforming to the original requirements of the software. Such costs can be either before (*internal*) or after (*external*) the software release.

The main assumption of the *PAF* model is that investing in conformance costs will pay off in reducing the cost of failure, and that investing in prevention activities will reduce appraisal costs [131]. In other words, there is an inverse relationship between conformance and non-conformance costs, therefore balancing this relationship between the two in a way to make it optimal is crucial to any software development process.

2. **Crosby Model**

   Similar to the *PAF* model, the Crosby model considers the cost of software quality as the price invested to conform to requirements and the price paid for non-conformance to requirements [131, 72]. The main concept of the Crosby model is that doing the job right the first time is cheaper than doing it later.

3. **Opportunity and Intangible Cost Model**

   This model was successfully integrated into the original PAF model by Sandoval-Chavez and Beruvides [130, 131]. The model quantifies the cost of losing opportunities and unrealised profit due to a reduction of customers and competition in markets. It argues that any lost profit that could have been achieved by increasing quality is to be considered as part of the total quality cost along with the conformance and non-conformance costs.

It would seem that a lot of features of the current CoSQ model are derived from or follow the same approach as the PAF model which divides the total cost into conformance and non-conformance costs. As is shown in Figure 3.2, which illustrates the relationship between the conformance and non-conformance costs, as the software quality increases by investing in appraisal and prevention costs, the non-conformance cost decreases. However, at some point during the software development process, the conformance cost increases rapidly, compared with a slight decrease in the non-conformance cost. The reason is that some errors may have a limited impact on the software compared to a significant loss in the time and effort required for finding and fixing them. A senior manager in one of the leading software development organisations stated that " I'd rather have it wrong than have it late, we can always fix it later." [116]. This supports the idea that balancing conformance and non-conformance costs is more cost-effective than trying to eliminate the non-conformance cost completely [117].

3.2.2 Industrial Data of CoSQ

In the previous section, a theoretical overview was given of the cost of software quality (CoSQ) and the models used to quantify it. This section presents some empirical data from the literature on the percentage that the cost of software quality represents out of the overall software development effort. According to a study conducted by the *Price Waterhouse Coopers* [123], which included a survey of 19 UK software development organisations, conformance cost (appraisal and prevention) make up 23-34% of the total software development cost. Moreover, non-conformance cost, that is the cost of internal and external failures, is weighted at about 41% of the total development cost. Another

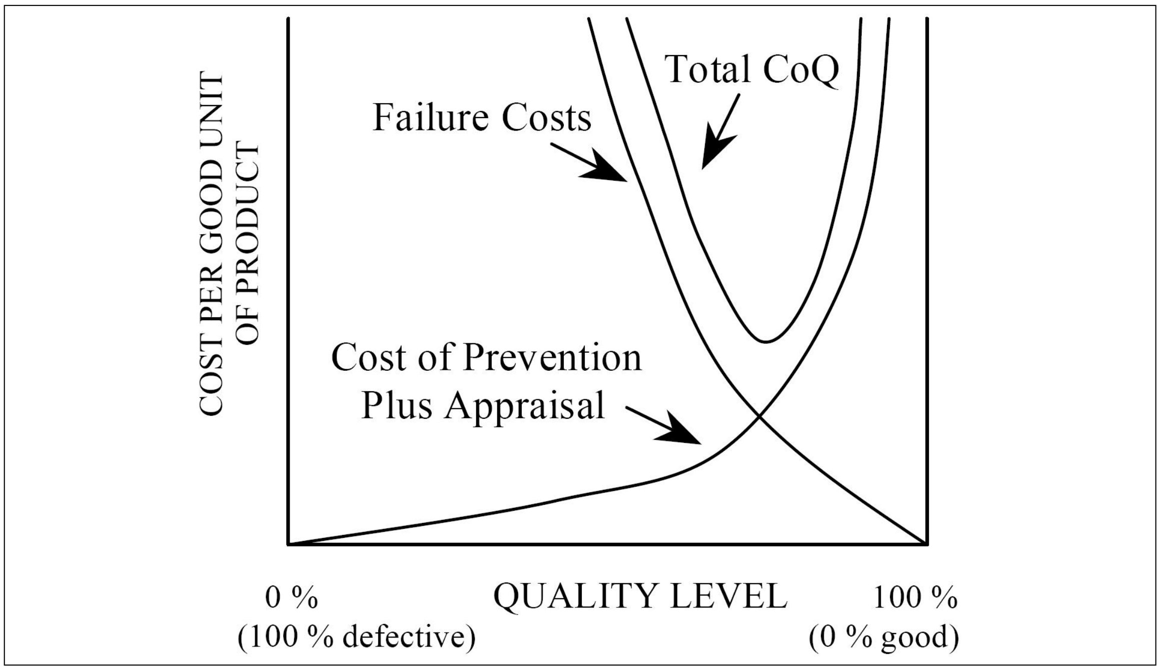


Figure 3.2: CoSQ Model : Source [131]

study showed that conformance cost consumes 40% of the overall cost of quality[134]. With regard to a specific quality technique, Votta [145], in his experiment, found that formal inspection, which is a typical defect prevention practice, consumes about 10% of the software development time.

3.3 Cost of Software Defects

It is generally accepted that during the software development process, the earlier a software defect is fixed and removed, the more the effort and time is saved for the whole project [78, 18]. Software quality researchers estimate that a software defect that cost less than $1 to be fixed during the software coding phase may cost $100 if it propagated to the whole system and thousands of dollars if it passed to the operational field. This is also supported by Humphrey [65] who showed that the rework cost of a defect found in the field is ten times greater than the cost to removing it during the system testing phase. In terms of actual cost amount, it is estimated that correcting a defect of a critical type in the coding phase is $977, this value would jump to $7,136 if this defect is corrected in the system testing phase [89]. The rationale behind this escalation in the cost of defects is that to fix a defect which was discovered relatively far from its origin entails not only the cost of fixing it but also the necessary re-working of other modules impacted by this defect in previous stages (Figure3.3).

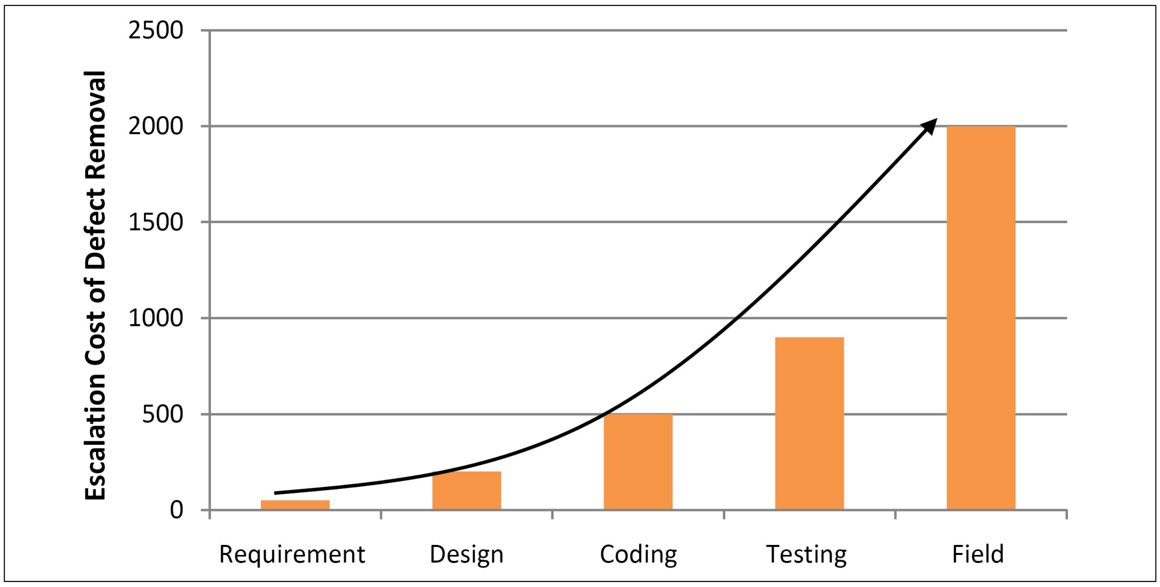


Figure 3.3: Escalation in Cost of Defects: *based on data available in* [102]

3.3.1 Cost of Rework

Rework is the process of revising a current artifact or deliverable to fix and prevent detected defects from recurring. Rework can be divided into two primary types of corrective work [45] :

- **Avoidable Rework**

  Avoidable rework is the work that could be avoided if the reworked artifact was correct, complete and in conformance with system requirements [45]. Accordingly, any effort spent fixing defects found during system testing which were injected from previous phases and could have been found there, is considered to be an avoidable rework cost [14].

- **Unavoidable Rework**

  Unavoidable rework results from unexpected changes to software requirements or any unpredicted software constraints [45].

  Software development organisations are more concerned with the cost introduced by the avoidable rework rather than the unavoidable rework's cost. The reason behind this is that the avoidable rework's cost can be controlled by implementing appropriate prevention techniques to reduce the cost of the rework which consumes a considerable share of the system development effort as mentioned before.

3.4 Cost, Quality and Time Relationship

In the software development process, project managers are always concerned about the software triple constraints which are: cost, schedule and quality [2]. This concern can be attributed to the inability of project managers to accurately quantify the trade-off between these three constraints. Many models have been developed in the past to address the optimisation of any constraint in isolation, keeping the other constraints as constant. However in reality there are trade-offs to be made. In some cases, the schedule becomes the most important aspect when the cost of delay outweighs the benefit of producing software of a low defect density. In other cases, a limited budget assigned to the quality plan hampers the quality assurance team from covering all software work products to detect and remove all defects. It is generally agreed that any software development project would experience the following scenarios:

- **Increase budget to meet quality and time constraints:** high quality and faster implementation require more investment.
- **Increase time to meet quality and budget constraints:** some less efficient tools and techniques requires more time but a smaller budget.
- **Decrease quality to meet budget and time constraints:** this is very common as high quality requires more investment than the other two constraints.

A well-known phrase among software quality practitioners is, "Too little testing is crime, but too much testing is a sin". This statement reflects the ongoing research problem in the software engineering area when project managers, at the start of or before the system testing phase, should make informed decisions to tune and control the triple constraints in a way to assure the success of their software projects. The triple constraints can be represented as the corners of a triangle as in Figure 3.4, where the software development process is in the middle of the triangle influenced by cost, quality and time constraints at each edge. The project manager has the choice to shift the software development process towards any aspect at the edges according to the software targets, constrains, budget etc.

For example, when a project manager wants to increase the quality level of the software development process, he/she would be influenced by the budget and time constraints which both need to be increased by different magnitude. Another option he/she may choose in order to isolate or reduce the effects of these constraints is to use quality tech-

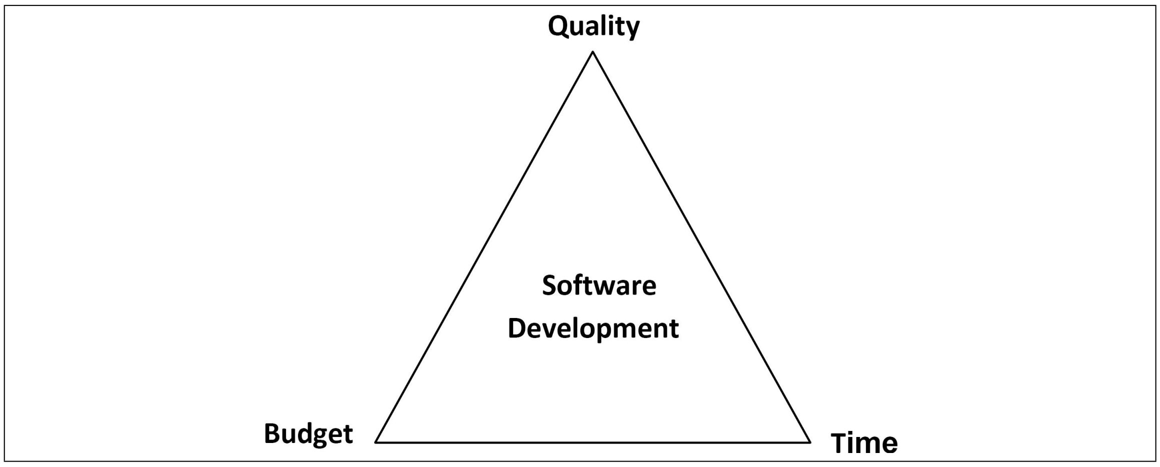


Figure 3.4: Software Triple Constraints

niques in a cost-effective way which is what this research tries to resolve.

As discussed before in Section 3.2.2, software quality entails a considerable amount of cost and may consume 50% of the total cost of the software development process [104]. Accordingly, investing in the quality constraint in a cost-effective way would result in a large saving in the total development cost which is represented by the budget and time constraints. In the following section, an outline is given of the state of the art software industry models that are currently being used for estimating software cost and quality and their pros/cons for dealing with the triple constraints problem.

3.5 Cost Estimation Models

The cost of software quality (CoSQ) relationship to the cost of production can be expressed by this simple equation:

Total development cost = Cost of production + Cost of quality

Assuming that a software development process has no or minimal requirements of quality, which is very rare, the total development cost would equal the cost of software development activities only according to this equation. Therefore, the two costs are to be well-estimated and quantified for better utilisation in regard to the triple constraints aspects and to establish accurate development plans and targets. In order to estimate the cost of production, software organisations use cost estimation models which help them determine the cost and schedule of their software development process. The following are the most common cost estimation models used by software engineers and project managers:

- Putnam's SLIM model
- PRICE Systems' PRICE-S model
- SEER SEM model
- Rubin 's and the Estimacs model
- COCOMO model

Most of these models are based on algorithmic equations derived from a regression analysis of hundreds of previous software projects.

The COCOMO model will be taken as an example to show how such cost estimation models work.

3.5.1 Constructive Cost Quality Model (COCOMO)

COCOMO, later COCOMO II, was published by Barry Boehm [16] as a schedule and cost estimation model. It is the most common and popular software cost and schedule estimation model used in the software industry. COCOMO uses the following algorithmic equation to calculate software development effort as a function of software size:

$$Effort = b.(KLOC)^c \tag{3.1}$$

- where *KLOC* is thousands of lines of code and b and c are constants depending on the type of software project under development. Those constants were derived from a regression analysis of 161 software projects.

In the simple form of the COCOMO model, there are three types of software projects:

- **Organic:** software projects developed by a small team in a known environment.

  b: 2.4 c: 1.05

- **Embedded:** software projects developed in an inflexible why and with strict constraints, i.e. defense systems.

  b: 3 c: 1.12

- **Semi-detached:** relatively large software projects with a team of mixed experience and skills.

  b: 3.6 c: 1.20

Having determined the type of software project from the above three types, software engineers enter the expected software size in thousand of lines of code KLOC and substitute the relative constants in the main COCOMO equation. Then, COCOMO will estimate the effort needed for the software development process on a person/month (*P/M*) scale. Moreover, based on the resulting effort value in person/month, COCOMO estimates the development duration using the following equation:

$$Duration = 2.5 * Effort^{c} \tag{3.2}$$

Similar to the effort equation, the *c* value depends on the type of project as follows:

Organic	0.38
Embedded	0.35
Semi-detached	0.32

Then the duration value is calculated as a function of effort to the power of the *c* value multiplied by 2.5.

As seen before, COCOMO is a cost estimation model which helps project managers make estimates of the effort and duration of current or future projects they may develop. Referring back to Equation 3.1 of total software cost, it can be seen that COCOMO will be responsible for the first part of the equation excluding the cost of the quality aspect (Figure 3.5).

In fact, COCOMO, to some extent, does include the cost of quality in the effort equation by using effort multipliers. The later version COCOMO II, incorporates several effort multipliers which were calibrated from several previous projects. These multipliers can affect the value of effort compared to the original COCOMO model. The new cost estimation equation for the new COCOMO II model is as follows:

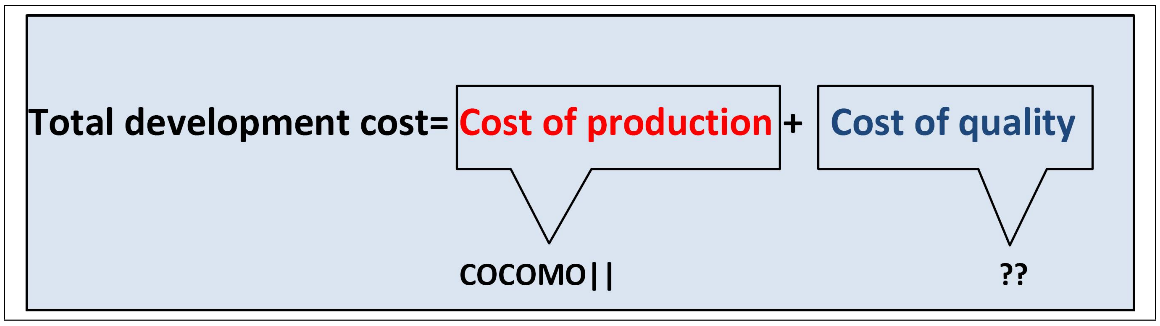
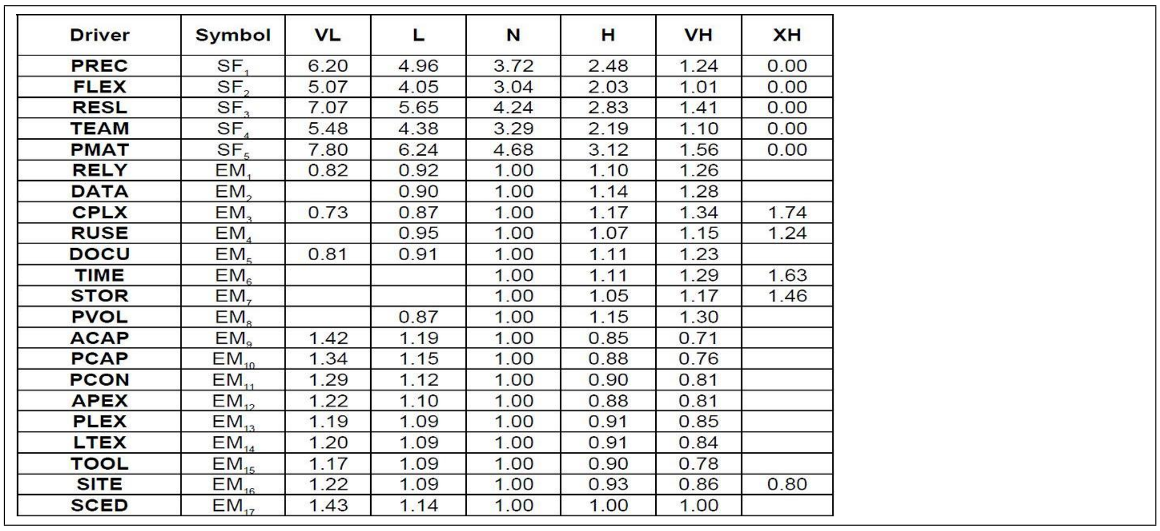


Figure 3.5: Software Development Cost

$$PM_{adjusted} = PM_{nominal} \times \left(\prod_{i}^{17} EM_i\right)$$

-where *PM_adjusted* is the adjusted person/month value after applying the required multipliers with the original person/month value. There are 17 effort multipliers and 5 scale factors, each of which takes a rating on a six-point scale ranging from very low to extra high, which in turn has an effect on the value of software effort. Those multiplier attributes are shown below in Figure 3.6.



Driver	Symbol	VL	L	N	H	VH	XH
PREC	SF_1	6.20	4.96	3.72	2.48	1.24	0.00
FLEX	SF_2	5.07	4.05	3.04	2.03	1.01	0.00
RESL	SF_3	7.07	5.65	4.24	2.83	1.41	0.00
TEAM	SF_4	5.48	4.38	3.29	2.19	1.10	0.00
PMAT	SF_5	7.80	6.24	4.68	3.12	1.56	0.00
RELY	EM_1	0.82	0.92	1.00	1.10	1.26	
DATA	EM_2		0.90	1.00	1.14	1.28	
CPLX	EM_3	0.73	0.87	1.00	1.17	1.34	1.74
RUSE	EM_4		0.95	1.00	1.07	1.15	1.24
DOCU	EM_5	0.81	0.91	1.00	1.11	1.23	
TIME	EM_6			1.00	1.11	1.29	1.63
STOR	EM_7			1.00	1.05	1.17	1.46
PVOL	EM_8		0.87	1.00	1.15	1.30	
ACAP	EM_9	1.42	1.19	1.00	0.85	0.71	
PCAP	EM_{10}	1.34	1.15	1.00	0.88	0.76	
PCON	EM_{11}	1.29	1.12	1.00	0.90	0.81	
APEX	EM_{12}	1.22	1.10	1.00	0.88	0.81	
PLEX	EM_{13}	1.19	1.09	1.00	0.91	0.85	
LTEX	EM_{14}	1.20	1.09	1.00	0.91	0.84	
TOOL	EM_{15}	1.17	1.09	1.00	0.90	0.78	
SITE	EM_{16}	1.22	1.09	1.00	0.93	0.86	0.80
SCED	EM_{17}	1.43	1.14	1.00	1.00	1.00	

Figure 3.6: Effort Multipliers and Scale Factors of COCOMO II : Source [12]

What links those multipliers to software quality is the **RELY** attribute which refers to the required reliability level a project manager seeks in the developed software. As an example, based on Figure 3.6, a very high reliability would increase software effort by 26% (*RELY-VH*= 1.26) which would be considered in th main effort equation.

3.6 Cost of Quality Models

3.6.1 COQUALMO

The constructive quality model COQUALMO is an extension to the cost estimation model COCOMO II [28]. It consists of two sub-models: 1, defect introduction and 2, defect removal models which in turn integrate into COCOMO as is shown in Figure 3.7.

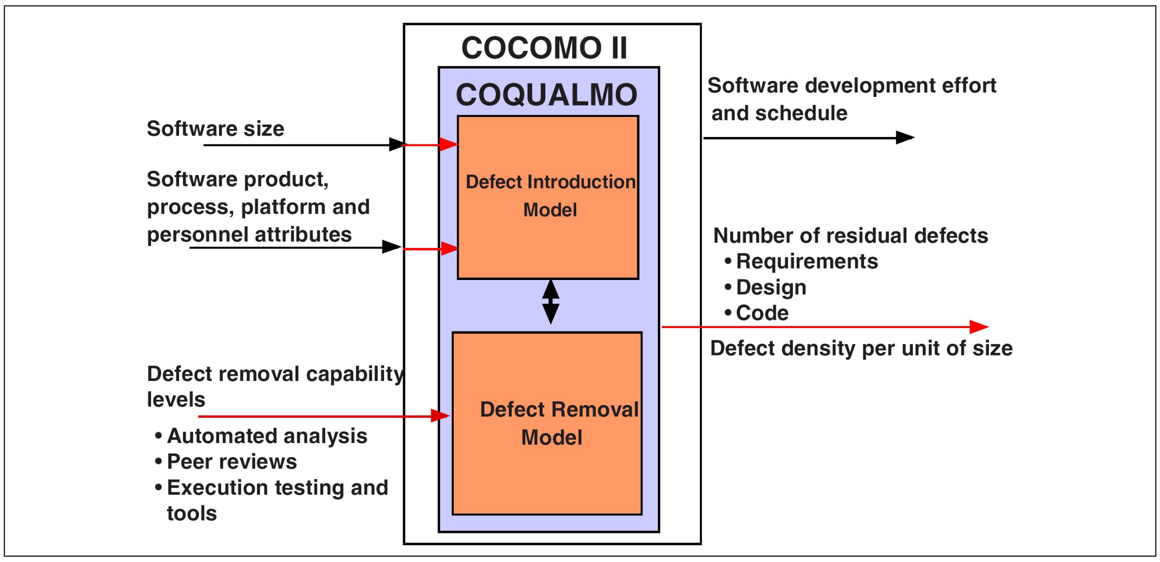

COCOMO II
COQUALMO
Software size
Software product, process, platform and personnel attributes
Defect Introduction Model
Defect removal capability levels
• Automated analysis
• Peer reviews
• Execution testing and tools
Defect Removal Model
Software development effort and schedule
Number of residual defects
• Requirements
• Design
• Code
Defect density per unit of size

Figure 3.7: COQUALMO as an Extension to COCOMO II : Source [98]

3.6.2 Defect Introduction Submodel

In his book, Barry Boehm [16] proposed a defect introduction and removal model based on the "*tank and pipe*" model proposed earlier by Capers Jones [80] (Figure 3.8).

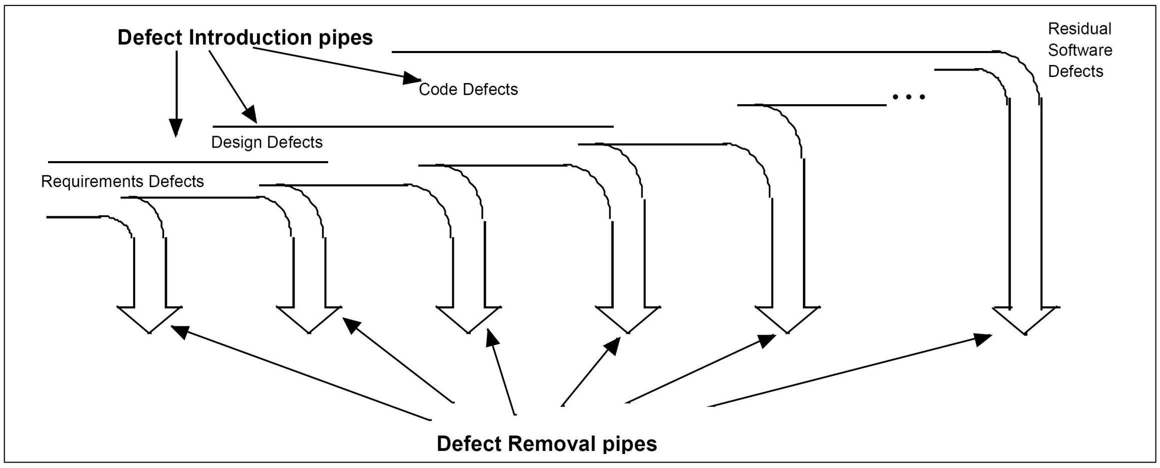


Figure 3.8: Pipe and Tank Defect Model: Source [28]

This model describes defect introduction and removal as a flow of pipes from each software development phase into a holding tank of defects at the end of the software. Introduction pipes are those through which defects reach the holding tank and form the defect introduction model, while removal pipes are those which drain the defects away and form the defect removal model. Defect introduction pipes represent the software development activities like requirements, design and coding which are part of the software development life cycle, whereas the removal model pipes include all QA defect removal activities and testing.

Similar to COCOMO II, the input of the defect introduction model is the estimated software size and a group of defect introduction rate multipliers. These multipliers are the same effort multipliers of COCOMO II discussed earlier, except for the FLEX (development flexibility) multiplier. This makes it easy to integrate the COQUALMO into the COCOMO of effort estimation as shown in the equation below. These multipliers affect the defect introduction rate based on their pre-determined numerical value.

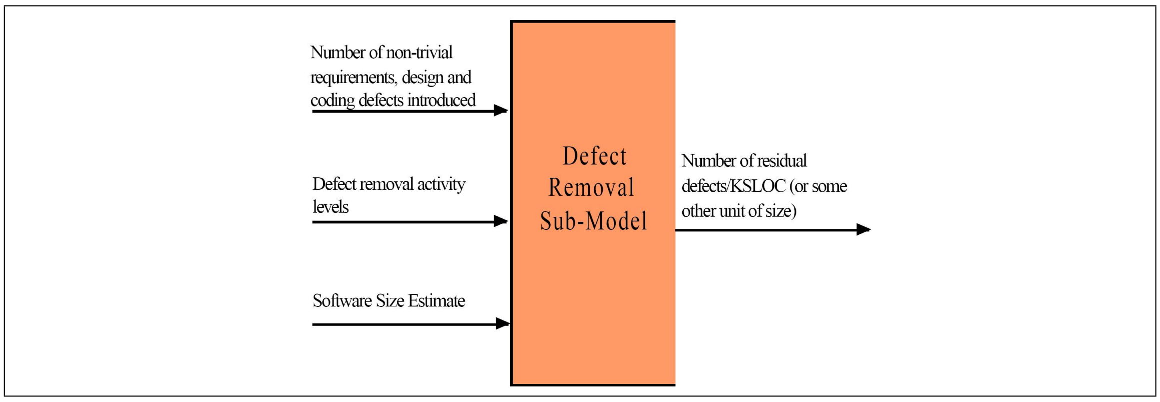
$$Number\ of\ introduced\ defects = \sum_{j=1}^{3} a_j * (Size)^b * \prod_{i=1}^{21} (DI - driver)_{ij}$$

- where:

- *j* represents the three phases: requirements, design and coding.
- *a* and *b* are calibration constants calculated from several previous projects as in the COCOMO equation.
- *(DI - driver)*$_{ij}$ is the defect driver that helps in introducing defects related to the *j* phase and the multiplicative factor **i**.

3.6.3 Defect Removal Submodel

The defect removal sub-model takes the input from the defect introduction model which is the estimated number of injected defects and then estimates the number of defects to be removed by applying three main defect removal techniques: automated analysis, review and testing.


Number of non-trivial requirements, design and coding defects introduced
Defect removal activity levels
Software Size Estimate
Defect Removal Sub-Model
Number of residual defects/KSLOC (or some other unit of size)
Figure 3.9: COQUALMO Sub-model

Each of those three techniques removes a fraction of requirement, design and coding defects according to the two-round DELPHI experiment conducted by the model author as shown in the equation below.

$$DRes_{Est.j} = C_j * DI_{Est.j} * \prod(1 - DRF_{ij})$$

3.6.4 Limitations of COQUALMO

Despite the fact that COQUALMO gives efficient and supportive software quality estimations, it was found that there are some limitations which affect the accuracy of the overall estimation result. For example, the effort to fix the defect introduced and removed is not quantified directly by the model. On the contrary, it is calculated as a percentage of the total estimated effort by the COCOMO model [148]. Moreover, defect removal techniques are limited to three: automated analysis, reviews and testing without taking into consideration the diversity of other techniques and the variations in their efficiency. It was shown in Section 2.5.2 that with a formal inspection, as an example, there are different reading techniques, each of which affect, negatively or positively, the defect removal efficiency of the formal inspection techniques, let alone the other QA practices. Moreover, COQUALMO does not associate time aspect to the defect removal and fixing processes which is a major issue in any QA process; instead it is integrated as a function of the COCOMO II duration estimation sub-model. Another limitation of COQUALMO is that defects are not given weights and classifications in terms of the software artifact they originated from. Even though there there have been some extensions to COQUALMO by categorising defects into high, medium and low categories or through the use of Orthogonal Defect Classification [98], issues pointed out in this research as discussed in Section 2.7 still have not been resolved.

3.7 Defect Prediction Models

3.7.1 Capture-recapture models

Transferred originally from biology, the capture-recapture principle is used in software engineering to predict the total number of defects in an artifact based on a sample taken from a population of defects [126]. The approach in biology simply says that to estimate the population of any animal sample what you need to do is to capture a set of that population, mark them and release them again. After a while, if you capture another set of the same population and when a marked animal is found it is considered as a recaptured animal and so on [19, 146]. Based on the number of recaptured animals biology researchers can estimate the total animal population of the sample under study.

The rationale for applying this principle to software engineering is that it follows the same process in testing and inspection. A tester takes a sample of defective software or module, fixes the defect and generates a defect removal report, and another tester discovers the same defect found previously and it is said to be a recaptured defect. Accordingly, the total number of defects can be estimated according to the equation below:

$$N = \frac{n_1 \times n_2}{m_2}$$

-where:

N: sample size to be estimated.

n_1: Captured defects on the first occasion.

n_2: Recaptured defects in the second occasion.

m_3: Marked defects recaptured on the second occasion.

However, this approach assumes that the sample of which the number of defects is estimated is a closed sample. In addition, the estimated number of defects have the same probability of capture which is not true as some defects are easier to be detected than others [149].

3.7.2 Growth Models

Software growth models are reliability measurement models that correlate defect data with an exponential function to predict how many software failures are expected to be encountered and to determine the best time to release software to users [153]. The defect detection rate in the testing phase is used by growth models to determine the defect density and reliability of the software. Growth models are dynamic and have a time component which helps determine some metrics like Mean Time Between Failure (MTBF) and Mean Time To Failure (MTTF).

An issue against growth models is that there are more than one hundred growth models available in the industry, each of which have huge variations and different characteristics [84]. This variety of models makes it difficult to choose a specific growth model that best fits an organisation. Another limitation is that such models are built on many assumptions which are usually violated in practice and may give inaccurate results [136]. Such assumptions are: faults having the same severity, using calendar time instead of execution time, etc.

3.7.3 Other Defect Estimation and Quality Cost Models

There are also other defect estimation models like Curve Fitting Models such as the detection profile [151] and cumulative methods [19]. Other recent models include using Bayesian Belief Networks (BBN) [23, 118], Neural Networks to estimate the number of defects in software modules [81, 56] and Fuzzy Classification[59]. These models predict the estimated number of defects injected into a module based on a sample set of defects reports during the software testing phase or by using prior information. Defect Prone Modules [74, 119] is also another approach used for estimating the number of software modules injected with defects. The classification process of modules is based on a set of metrics and attributes that are meant to contribute to the increase of defect injection in a module. Such metrics are Cyclomatic Complexity and other object-oriented metrics [74, 24, 113]. However, defining fault prone modules relying on such code-based metrics is a very late stage in the software development process. Accordingly, cost invested in the QA practices in the early phases of the software is not covered by this classification.

3.7.4 Discussion

Despite the proven effectiveness of the models discussed earlier in defects estimations and saving quality cost, there is still an absence of a holistic quality model that encompasses all phases of the software life cycle and helps in controlling the investment of QA practices in those phases. Moreover, the models discussed earlier work in isolation for a specific purpose without taking into account the association of the triple constraints of the software life cycle. Such models do not have the ability to provide an informed decisions regarding QA plans based on the aspect of cost effectiveness . In addition, the risk-based aspect of specific modules is neglected in the classification-based models where the aspect of what is defect-prone, as an example, is the main classification criteria.

3.8 Relationship Between ROI and QA Activities

As was discussed before, the time when quality was considered as a final step in the software development process has passed because of the considerable effect quality has on the overall cost and hence on the expected profit [92]. The most well-known business measure that computes how efficient a business investment is in gaining profit is the Return On Investment (ROI) (Figure 3.10).

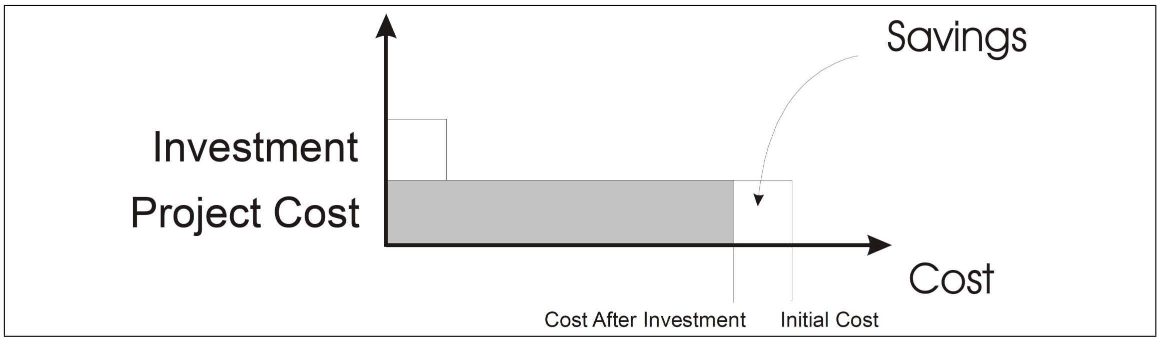


Figure 3.10: ROI Concept : Source: [41]

Many studies have shown the relationship between software quality in terms of the verification and validation techniques and the ROI principle [78, 84]. In general, the factors that ROI incorporates are the time benefit and cost benefit excluding the unquantifiable benefits like customer satisfaction, higher position in the marketing , safety etc. As was earlier shown in Section 3.4, Implementing quality techniques and practices does affect the time aspects and the total development cost aspect. Accordingly, those two aspects should be well-identified in order to get an accurate ROI value for any decision on quality [92].

As shown in **Figure 3.11**, the relationship between software quality and profit has

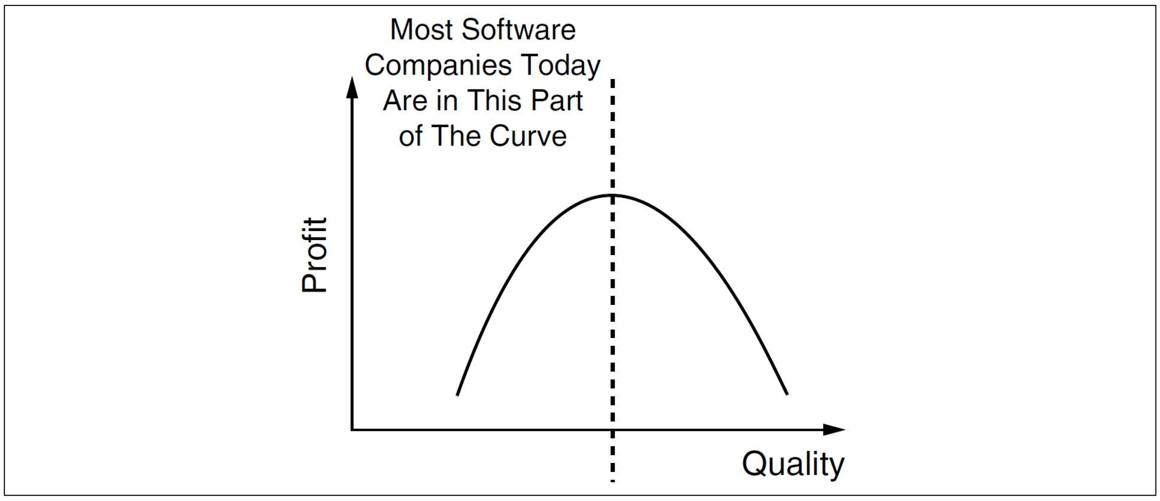


Figure 3.11: Relationship Between Quality and Profit : Source:[41]

a linear-based relationship at the beginning, as implementing practices and techniques removes defects and improves software quality. However, this linear relationship reverses at some point during the software development process with a decrease in expected profit and a continuous increase in quality. In his book, Khaled El Emam [41] argues that most software companies stand on one point of this curve without reaching the optimal point shown by the dotted line.

3.9 Pareto Principle

The Pareto principle or what is known as the 80 : 20 rule was created by the Italian economist Vilfredo Pareto in 1906. This rule argues that wealth distribution in Italy was distributed according to 80 : 20 rule, so that 80% of the country's wealth belonged to 20% of the people. Pareto's law was evaluated by many researchers in order to test its validity, and suprisingly they found that this phenomenon was applicable to different areas of research which can make it a natural law where for anything in life, 80% of effects come from 20% of causes. McCabe applied Pareto's Law to software quality activities by arguing that 80% of software defects belong to 20% of the code [100]. Moreover, Barry Boehm found that it is not only software quality that conforms to Pareto's Law as stated by McCabe, but also that 20% of code consumes 80% of software resources, maintenance costs and execution time [13, 94]. Ostrand and Weyuker analysed a considerably large system of 500 KLOC of 13 different releases and found that defect distribution conformed to Pareto's Law [111]. However, this law is given to software quality personnel theoretically with the absence of a model which can utilise it effectively during the software testing stage. Also, researchers who looked at Pareto's Law from a software defect perspective restricted it to the system testing phase, overlooking defects created during the whole software development life cycle where the cost of QA practices and techniques are considerably high and consume a large stake of the software budget and time.

From a software development perspective, occasionally the process of developing software deliverables like requirements, design and code may be shortened due to time constraints. For example, requirements document of 500 Functional Points (FP) may not be fully inspected as the time to release constraint exerts pressure on the project manager. Moreover, the project manager may decide to re-use a module from a previous release in order to shorten the development process time and to reduce the development budget(Ariane 501 Section 1.1). However, in the first scenario, what is the criteria behind the decision of leaving work products uninspected or re-using a module; how can the project manager handle such issues with minimal risk?

3.10 Summary

This chapter reviewed the cost of software quality (CoSQ) principle and the major role it represents in the decision-making process for QA activities. It showed based on literature real data resulting from empirical studies the percentage of cost that the cost of software quality consumes from the total software development budget. Moreover, the software triple constraints: cost, quality and budget, which constrain the software development process were outlined. Upon our review of the state of the art models there was an absence of a well-defined model which can make the trade-off process between those constraints in such a way that gives the project manager the ability to contrast and compare potential QA plans. Examples of currently used models for CoSQ estimation were outlined as well as their drawbacks in terms of dealing with the software triple constraints problem. From a financial perspective, the relationship between the ROI metric and the CoSQ principle was discussed and how profit, represented by the ROI metric, can be used along with the triple constraints to evaluate QA plans in terms of reducing the waste spent on unnecessary QA improvement. Finally, an overview to the Pareto principle was given which states that 20% of causes are repsponsible for 80% of effects. It showed how this law can be applied to the software QA activities as 80% of software defects originate from 20% of its artifcats.

Generally there are four main shortcomings upon our revision of the earlier sections:

- The current software cost estimation models based on the manipulation of a pool of data of different types of software projects which make them too generic to be applied to all organisations which reduce their desired accuracy.
- The cost of software defects is estimated generally without taking into account the

different types of defects, their severity levels and their relative locations in the software development process.

- There is an absence of risk factor consideration in the QA activities for the software development process. There must be a mechanism that gives the project manager the ability to differentiate between high-risk modules that require more investment in QA activities and low-risk modules.
- The Pareto principle which is based on empirical observation states that 80% of effects are related to 20% of causes; this can be generalised in software by stating that 80% of software defects originate from 20% of the code.

Other shortcomings covered in the previous chapter are:

- *The traceability problem of the efficiency of a QA practice in the part of the software it is applied to.*
- *The DRE value is generalised and does not account for the variation of defect types and severities.*

Chapter 4

Software Quality Management Model

Objectives

- To propose our dynamic QA model.
- To determine the components of the model and how they interrelate with each other.
- To measure the effects of choosing one specific QA practice over another.
- To facilitate the process of counting the number of defects detected by QA practices.
- To calculate the cost of execution of using QA practices.
- To calculate the execution duration needed to run QA practices.
- To determine the return on investment (ROI) metric and how it can be integrated within our model.

4.1 Software Quality Framework

This chapter proposes our software quality management model which will try to resolve the research questions pointed out in Chapters 2 and 3. As pointed out that there are some shortcomings in the literature in terms of software QA activities that need better consideration by software QA researchers. Such shortcomings are very important to QA practitioners and have crucial effects on the efficiency of any QA plan. With regard to the software triple constraints: quality, cost and time, it seems that the latter two aspects can be explicitly quantified using (man/hours) as a unit of money for the cost constraint and clock time for the time constraint.

In terms of the quality constraints, as discussed earlier in Section 2.1.1, quality can be defined and measured based on different perspectives: conforming to requirements, value-based, etc., and defined according to the extent of conformance to the well-known process quality models like *CMMI*, *ISO 9001*, and the product quality models like McCall, Boehm, etc.

4.1.1 Quality Metric

Our model is built upon the fact that the quality of software is intrinsically linked to the notion of software defects and defect density. Therefore, our model relies on the number of defects injected into a software as a metric for defining the first aspect of the triple constraints, which is quality. This metric can be re-defined by the defect density metric or in other words, the number of defects injected into a software with a unit of size.

$$\boxed{Defect\ density = \frac{total\ number\ of\ defects}{software\ size}}$$

Using this metric would help in overcoming the triple constraints trade-off problem pointed out in the research question as there would be a dynamic trade-off process that would include all the three constraints. Many models have been developed in the past to address the optimisation of two of the three constraint in isolation, keeping the third constraints as a constant. However, in reality, there are trade-offs to be made. In some cases, the schedule (ie. development time) becomes the most important constraint, when cost of delay outweighs the benefit of producing software of a high quality (i.e. low defect density). In other cases, a limited budget assigned to the quality plan hampers the quality assurance team from covering software work products sufficiently to detect and remove defects, even those that are deemed to be of a high or moderate risk. These relationships are depicted to help to qualitatively make the trade-off process in Figure 4.1.

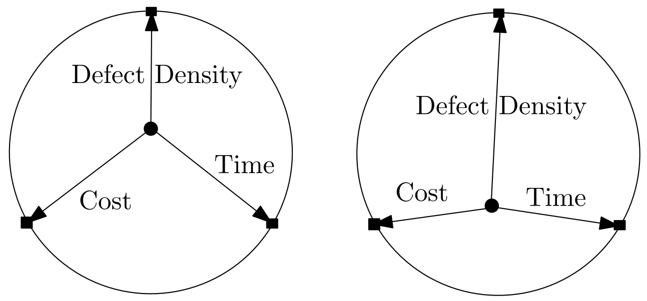


Figure 4.1: Trade-off: Cost, Quality and Time

The Defect density is expressed in Figure 4.1 as being inversely proportional to quality. A high quality product exhibits a low defect density. Referring to Figure 4.1 a project manager has the freedom to navigate the project between the triple constraints. When trying to reduce the cost (moving from the left scenario to the right), he/she is likely to reduce the quality as less money would be available for defect detection and verification activities. This may also reduce the length of time that can be spent on the project. Similarly, a reduction in the time for the project can lead to higher costs through additional employees and/or to lower quality products due to the omission of quality assurance mechanisms.

4.1.2 Risk-based Approach of Software QA Activities

Following the *CoSQ* approach and the Pareto principle discussed in Sections 3.2 and 3.9 respectively, a novel approach is re-defined by which investment assigned to QA activities can be utilised in a cost-effective way which achieve and fulfill the previously mentioned aspects. As mentioned in Section 3.9 that 80% of injected software defects originats or are related to 20% of the software. In that context, QA activities should be wisely and selectively applied for a better defect detection and removal process. Moreover, risk considerations should be well taken into account during the software development process.

Software artifacts vary in their importance and their risk impact on the development process and to the end-user, which introduces the need to have a categorisation mechanism of software artefact and software modules by which software project managers can prioritise the investment of the QA activities they have conducted. For that purpose, as will be shown later, our model categorises deliverables of the SDLC phases by giving them different weights of consideration and priority. This allows software project managers to control the available budget assigned to quality assurance plans according to the risk level of each of the phase's deliverables and choose and deploy the suitable verification and validation activities accordingly.

Our model consists of two main parts:

- The regression analysis which encompasses the data collection and analysis process.
- The mathematical formulas which present formal equations to utilise during data analysis process.

4.2 Framework of the Quality Model

Our approach to the optimal use of quality assurance practices and performing the required trade-off process between the software triple constraints is regression-based. In other words, a regression analysis is performed on a pool of QA data which was grouped and channelled according to our model specifications. The output of this analysis process will help determine the accurate effectiveness of QA practices, defect detection time and removal cost with respect to categorised software work products on the basis of the risk level associated with them.

The effectiveness is therefore reliant on the data collection of previous projects. In Figure 4.2, the different activities that are involved during project planning is shown and how our model is used by project managers to manage project quality assurance plans.

Firstly the project is broken down into phases, depending on the life-cycle model that is used. For the waterfall model, project phases can be determined upfront, whereas for more agile methodologies the planning needs to be on a per phase basis. For each phase, there should be clear conditions on the inputs of that phase. Typically the inputs of a phase are the same as the outputs of the phase's predecessor. The outputs of a phase are captured in the work products that are developed or refined in that phase. Conditions of the output could be functional or quality related. In our model, a development phase is

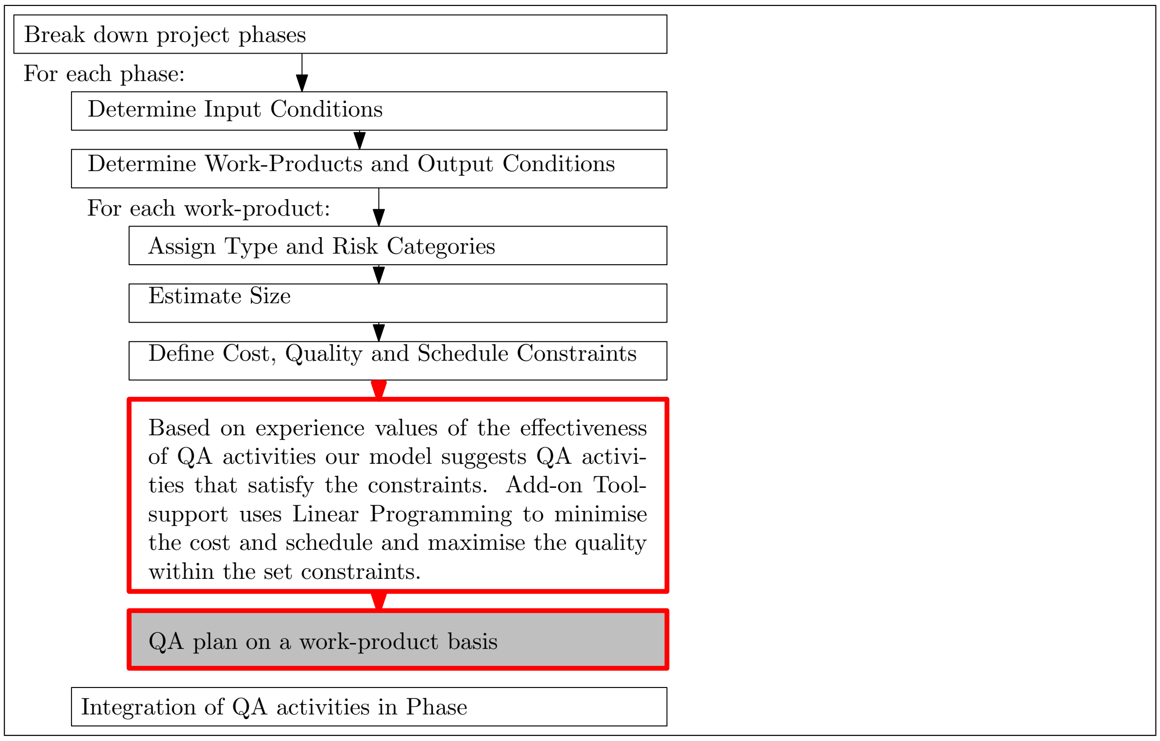

Break down project phases
For each phase:
Determine Input Conditions
Determine Work-Products and Output Conditions
For each work-product:
Assign Type and Risk Categories
Estimate Size
Define Cost, Quality and Schedule Constraints
Based on experience values of the effectiveness of QA activities our model suggests QA activities that satisfy the constraints. Add-on Tool-support uses Linear Programming to minimise the cost and schedule and maximise the quality within the set constraints.
QA plan on a work-product basis
Integration of QA activities in Phase

Figure 4.2: Project Planning Work-flow

assumed to generate a set of work products. These work products can be of different types and can carry various levels of risk.

To enable a more accurate regression of software projects, project managers are allowed to place work products into type and risk categories, with the assumption that cost, quality and time measures within each category are correlated over the projects. Following the categorisation of type and risk, the size of the work product should be estimated. This can be done using well-established effort estimation models like COCOMO II or simpler approaches like design/code expansion ratios. The constraints on budget, schedule and quality for the overall project should be broken down into constraints on the individual work products. In our model, a variation of the defect depletion model is proposed that allows the capture of the impact of defects present in the phase's input on the quality of the work product generated in this phase. After the constraints on the work products are determined, our model uses regression data from previous projects to estimate the likely defect density of the work product based on its size and type. A linear programming techniques is then be used to optimise the application of QA practices available within the organisation based on their effectiveness, cost and impact on schedule. This takes into account the effectiveness of the QA practices with regard to the size and type of the work product and may result in a combination of QA practices being suggested. The selected activities are then integrated into the development plan for the phase.

4.2.1 Work-flow of the Quality Model

The application of our model conforms to any software development life cycle as long as it consists of phases as in Figure 4.3. The waterfall model as discussed in Section 2.3 was taken as an example of a software life cycle model. As were shown this type of life cycle model consists of a number of development phases which rely on the deliverables of each other, and lastly that there is a system testing phase where the main testing activities of the software occurs. In our model, each development phase in the software life cycle is associated with the system testing phase with respect to the defects detection and removal activities. This association does not restrict the use of our model on a per phase basis, It can be extended to associate the other phases with the system testing phase or to include the association between the phases themselves. This subsection outlines the work flow steps of applying our proposed model for each software development phase.

During the development process, the deliverables of each phase will go through a categorisation process to categorise the artifact of each phase into different work products; the mechanism of this categorisation process is described later. As discussed in the introduction of this chapter, the risk-based approach of our quality model demands that phase deliverables need to be dealt with differently in order to prioritise the investment available for QA plans. The next step after the categorisation process is the constraint check, which is a decision-based process conducted by either the system's project manager or the system's stakeholders who may raise some issues and terms to be considered in the system to be built. For example, such a constraint may include but is not limited to a desired value of defect removal efficiency for single or total work products of the phase. In other words, the project manager may require a *DRE* of 80% as a success target of a specific QA process, that is, that 80% of the estimated defects would be detected and fixed. Another constraint could be a specific level of cost invested in QA activities. Moreover, in any software development project a repetitively occurring constraint is quicker software release time due to unexpected conditions. Accordingly, a constraint check step is designated.

Once constraints have been determined they are put through the optimal solution determination process. In this process, constraints are grouped together and processed using a suitable optimisation technique which will propose an optimal solution for the QA plan. The optimal solutions found will be based on two or more of the software triple constraints: cost, quality and effort. In the case there is no optimal solution found, the constraints should be re-discussed between the software stakeholders and the project manager, and re-designed for better manipulation of the optimisation process. Once optimal solutions are found or if there are no constraints determined, the QA team should start the QA process by using the suitable QA practices.

The data of every QA practice in terms of the number of defects found, the duration of executing the QA activity, the average defect removal cost of the QA practice, etc., will be passed to the defect repository where it is channelled and stored according to the

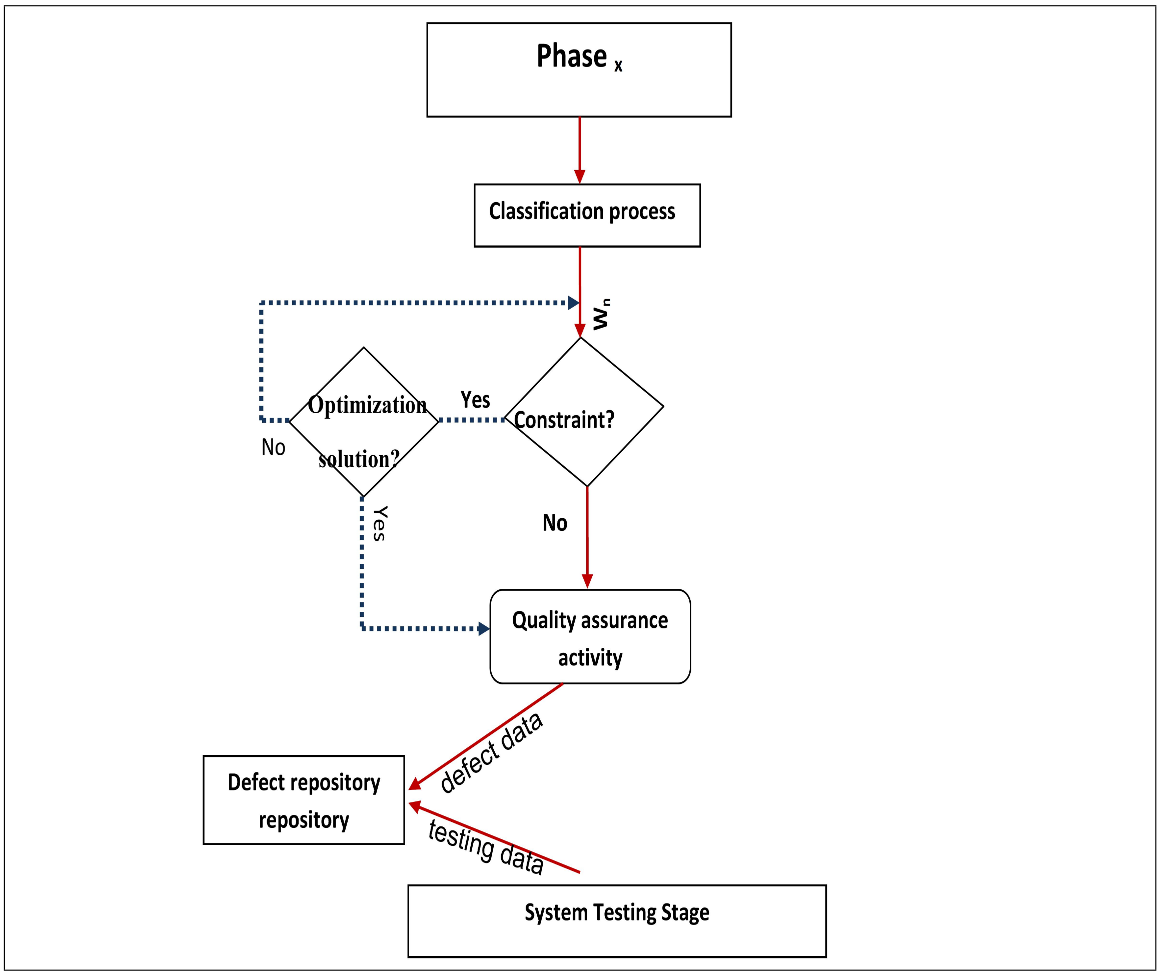


Figure 4.3: Work-flow of the Quality Model

categorised work products. This approach is applied to each phase, following the same procedures and conditions.

In the system testing phase, where the model terminates, the main testing activities will uncover more defects that have escaped from each phase and from each classified part of the phase. Then the data of defects uncovered from each work product and the defect removal efficiency *(DRE)* value assigned to each QA practice with respect to that work product will be updated in the phase repository to be used in regression analysis for future use. So, there is a matching process between the main development phases; *(Requirement, Design, Coding*) and the system testing phases.

4.3 Risk Association and Software Phases

In our approach, a more accurately decomposed QA process is proposed. As discussed earlier, the software project manager should invest the available budget assigned to quality very carefully and with a pre-defined scope in terms of assigning targets and goals rather than one single goal of eliminating or alleviating the impact of defects. It is assumed in this research that software development organisations should have a list of risk rating levels that are considered differently with regard to their significance, the impact level of their defects. Moreover, with regard to their priorities and investment paid to the QA process that deals with each one of them. A graphical overview of the risk consideration of software is shown in Figures 4.4 and 4.5.

Therefore, the deliverables of any software development phase: software requirement specification *(SRS)*, high level design document, low level design document, code etc., should be classified according to those risk levels. Consequently, for a QA plan of a

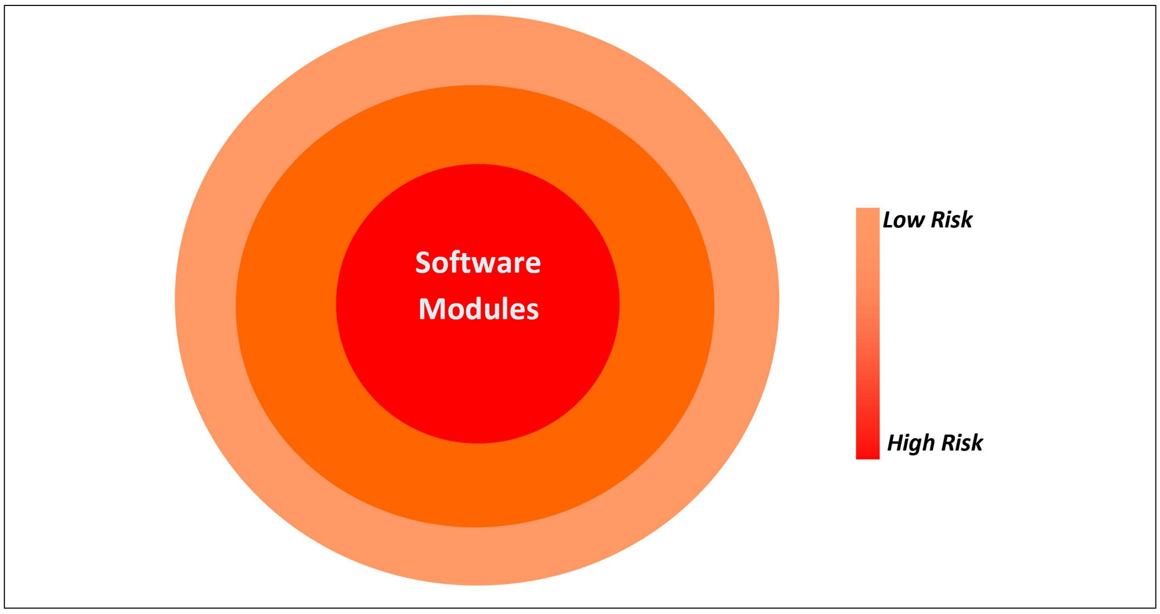


Figure 4.4: Software Risk Levels

limited budget, the highest share of the budget should go to the most critical levels rather than distributing it evenly across the whole phase. For example, in an *SRS* document which contains a description of the core modules of software to be built, there will be extra pages to describe the non-functional aspects related to some modules with regard to their security features, response time, dependability etc. Taking the domain of the software into consideration and with a worrying constraint such as a limited budget, the first priority for the QA team would be to inspect the functional requirements of the modules before moving towards the non-functional aspects of the modules.

The categorisation process of software modules to fulfill the risk consideration proposed in this research is straightforward for some artifacts and somewhat complex for others. The reason for such complexity is due to the cohesion links and compositionality levels within each artifact itself. The software requirement specifications (SRS) is taken as an example for the sake of simplicity and clarity. This methodology is applied to the software requirements specifications *(SRS)* in a concise way so that the idea behind the

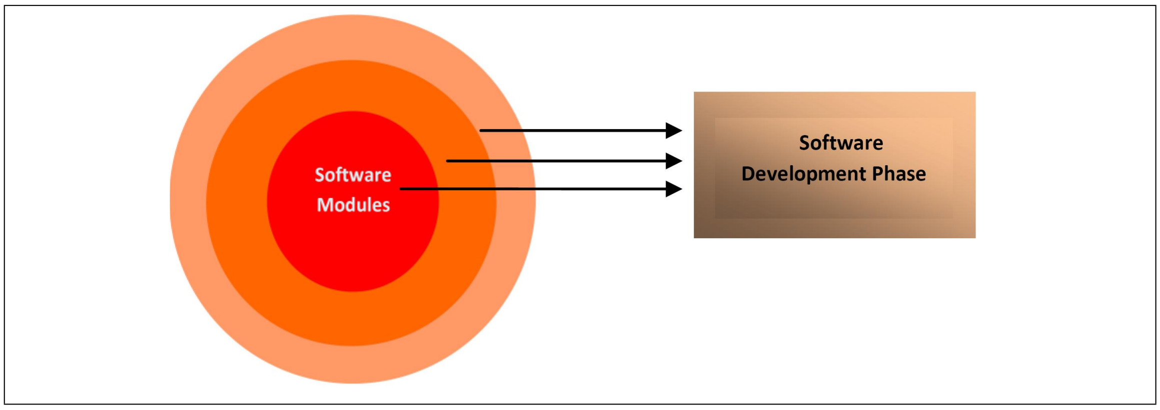


Figure 4.5: Risk Levels and Phases Association

approach can be fully comprehended.

4.4 Development Phases Categorisation Process

The project deliverables of each phase in the SDLC can be broken down into work products. The assumption in this research is that these work products can be assigned to a domain of a specific type and risk category. For a limited QA planning budget, the highest share of the budget should go to the most critical work products rather than being distributed evenly. For example, in a software requirements specification document, requirements are typically ranked based on importance to the client. Here, more effort should be given to the verification of requirements that are important to the user than to those that are in the "waiting room"[128]. Having determined the risk levels, each work product of each of the software development phases is classified based on a defined set of risk levels derived from the consideration of the risk associated with the software development process.

This section shows the categorisation scheme of work products followed in this research.

Let C be a set of all categories that are used in the domain for which the software is being developed. Every work product can be placed in one category. Let W_x be the set of work products that are generated in phase x. The function $type : W_x \mapsto C$ then assigns a type category for every work product. Multiple categorisation schemes can be modelled along these lines. Similarly let $esize : W_x \mapsto N$ be a function mapping from a work product to an estimated size in KLOC. Although for early work products other size measures, such as object or function points, are more suitable, it is well established to translate these into KLOC for the target language of the project [16].

The size $size_x$ of the output of phase$_x$ is then

$$size_x = \sum_{w \in W_x} esize(w)$$

The size of work products of a specific type $c \in C$ is then:

$$size_{x,c} = \sum_{w \in W_{x,c}} esize(w)$$

where $W_{x,c} = \{w \in W_x | type(w) = c\}$.

The proportion of work products with what is in category c is then:

$$\alpha_{x,c} = \frac{size_x}{size_{x,c}}$$

The categorisation of the phase's deliverable work products depends on the insight of the project manager or the QA team's consideration for the deliverable itself and for the type and prospecting domain of use of the software. Some deliverables would be given a full weight value $\alpha_{x,high}$ of a high-risk rating out of the total deliverable size $size_x$. Conversely, others will be given a full weight value $\alpha_{x,low}$ of a low-risk rating. An example

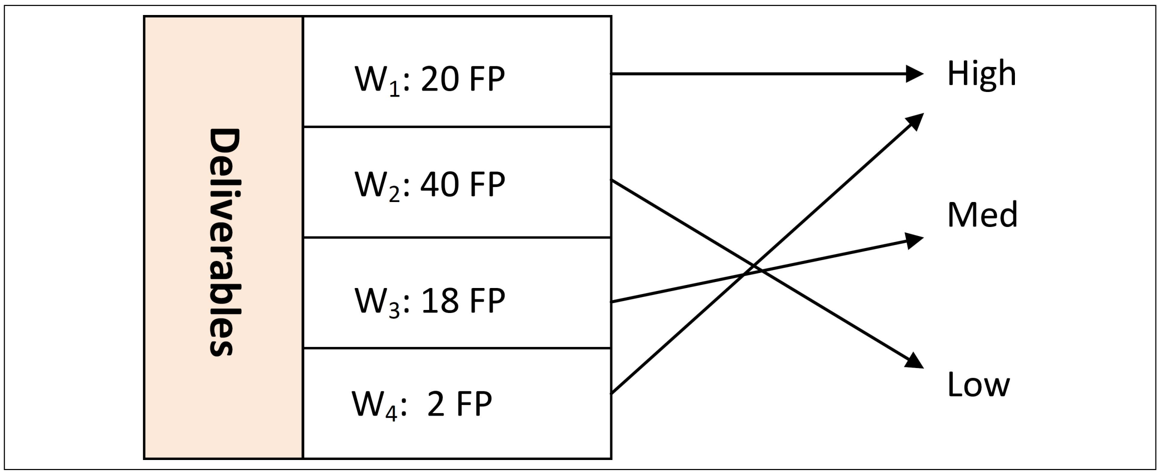


Figure 4.6: Phase Categorisation Process

of a mapping from work products to risk categories is shown in Figure 4.6.

4.5 Categorisation Scenarios

The categorisation process of software phases into work products is independent of the software size and the risk levels defined. That is, having a set of risk levels does not mean that each phase deliverable is categorised according to these levels at once. The constraints experienced during the software development process and the project manger's insight are the main determining factors for the way this categorisation process is implemented. The following subsections will show how this methodology is applied to the software requirement specifications by giving some scenarios of work products categorisation.

4.5.1 Scenario 1 :

As an example and as illustrated in Figure 4.7, the 1st scenario shows that SRS document is divided according to our risk-based levels (*high, medium, low*) into three work products of sizes 30%, 50% and 20%, which are inspected by QA practices P_1, P_2 and P_3 respectively.

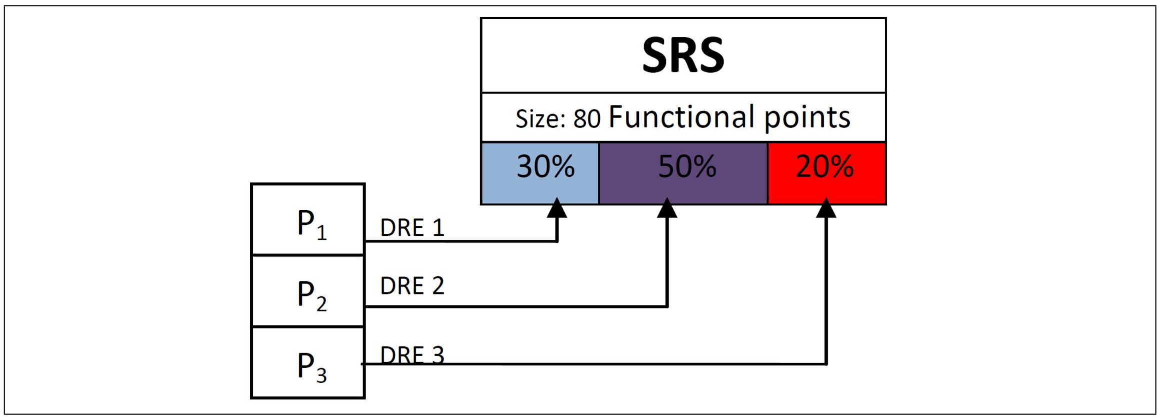


Figure 4.7: Work Product Categorisation (a)

4.5.2 Scenario 2 :

In another scenario shown in Figure 4.8, the project manager may have no available budget to inspect all the artifacts or some of the QA practices may only be available for a limited period of time due to the fact that the testers who are used to this technique are busy or not available or due to any other reason. Accordingly, the QA team manager would choose to inspect the high and medium work products only, which are weighted at 50% of the total artifact with the coverage values of 30% and 20% for QA practices P_1 and P_2 respectively. The remaining 50% of low-risk work products would be left uninspected.

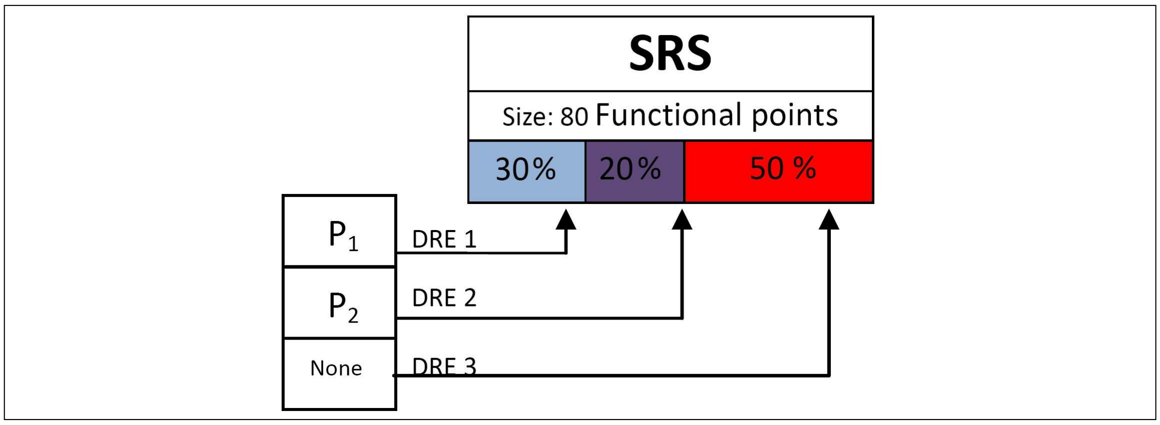


Figure 4.8: Work Product Categorisation (b)

4.5.3 Scenario 3 :

Given a software project of high criticality, the project manager tells the QA team to consider the software project carefully. The QA team divides the SRS artifact into two work products: a high work product with a weight of 80% and a medium work product with a weight of 20%, and both work products would be tested with QA practices P_1 and P_4 (Figure 4.9).

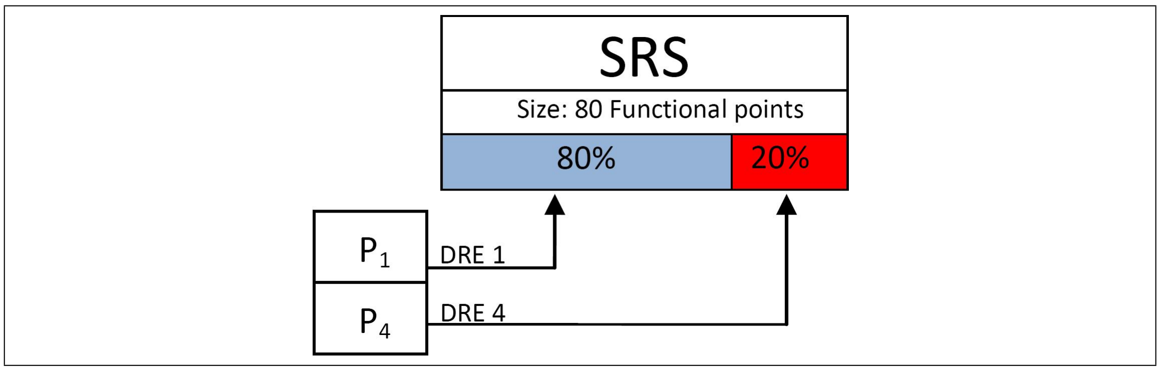


Figure 4.9: Work Product Categorisation (c)

4.5.4 Scenario 4 :

Moreover, the categorisation process may continue to include the work product itself with respect to the coverage weight given to the QA practices chosen. For example, in Figure 4.10, the project manager may not give practice P_1 full coverage on the work product of a high type because the availability of P_1 could stop all of a sudden in the middle of the testing or due to another constraint. Accordingly, the project manager would use practice P_3 to inspect the remaining part of the work product.

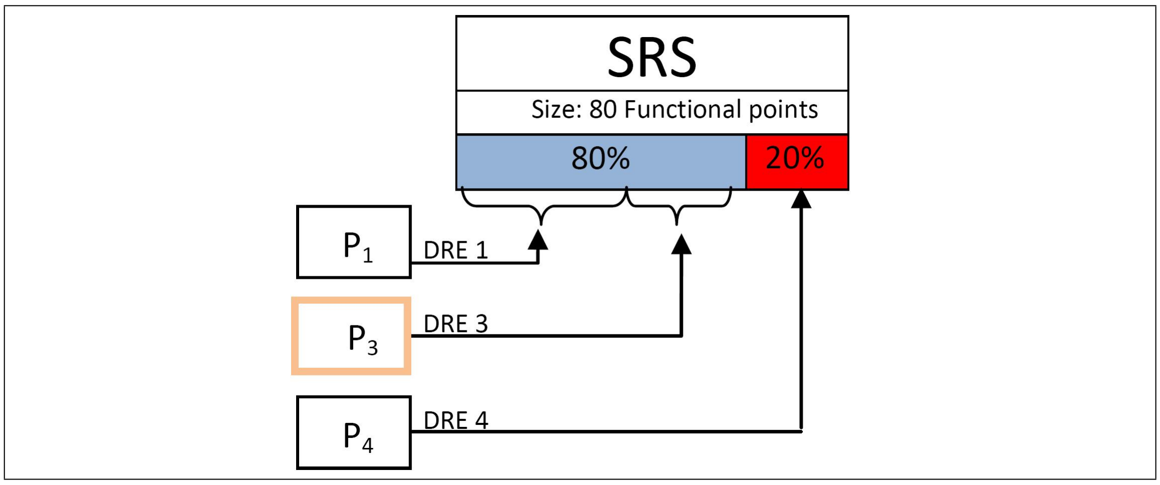


Figure 4.10: Work Product Categorisation (d)

This assignment mechanism occurs commonly in the industry without prior planning due to the various unexpected constraints and obstacles that a QA team may experience which would compel them to use whatever QA practices are available.

The four different scenarios mentioned before occur repetitively in a software development environment without being exploited for future use. For example, let us imagine that the second scenario happened again and that the project manager should leave part of the artifact uninspected to release the deliverables quicker. How can the project manager can use his/her prior knowledge to make an informed decision on such a scenario and what the consequences are likely to be ?. Accordingly, software organisation should have a comprehensive and solid background in the efficiency of their QA practices for each phase and should also have a QA repository of previous projects in order to manipulate it for future use. The following sections, will propose the supportive tools and techniques that will help in building up the QA repository on the basis of the categorised work products presented.

4.6 Advanced Defect Containment Matrix

Referring to our discussion in Section 2.6.1, the defect containment matrix, which is currently used to assess the efficiency of QA activities, calculates the defect removal efficiency (DRE) value for the whole phase, that is, it measures the success of the whole QA activity. In that respect, it fails to accurately define the DRE value of a specific QA technique in the case of more than one QA technique being applied within one phase as the DRE given by this matrix is general to the phase. Moreover, it failed to quantify the variation of defects in terms of their impact severity by illogically considering the whole phase as being equal.

In order to reduce the weaknesses of the original defect containment matrix and in order to support the risk-based approach mentioned in our framework, an advanced defect containment matrix is proposed. A graphical depiction of the matrix is shown in Figure 4.11. The initiation of this matrix is at the beginning of the development activities for every phase. The software to be developed will utilise the proposed matrix after the categorisation process so that each phase is categorised into different work products. Once the QA activities start, each categorised work product is assigned one or more QA practices.

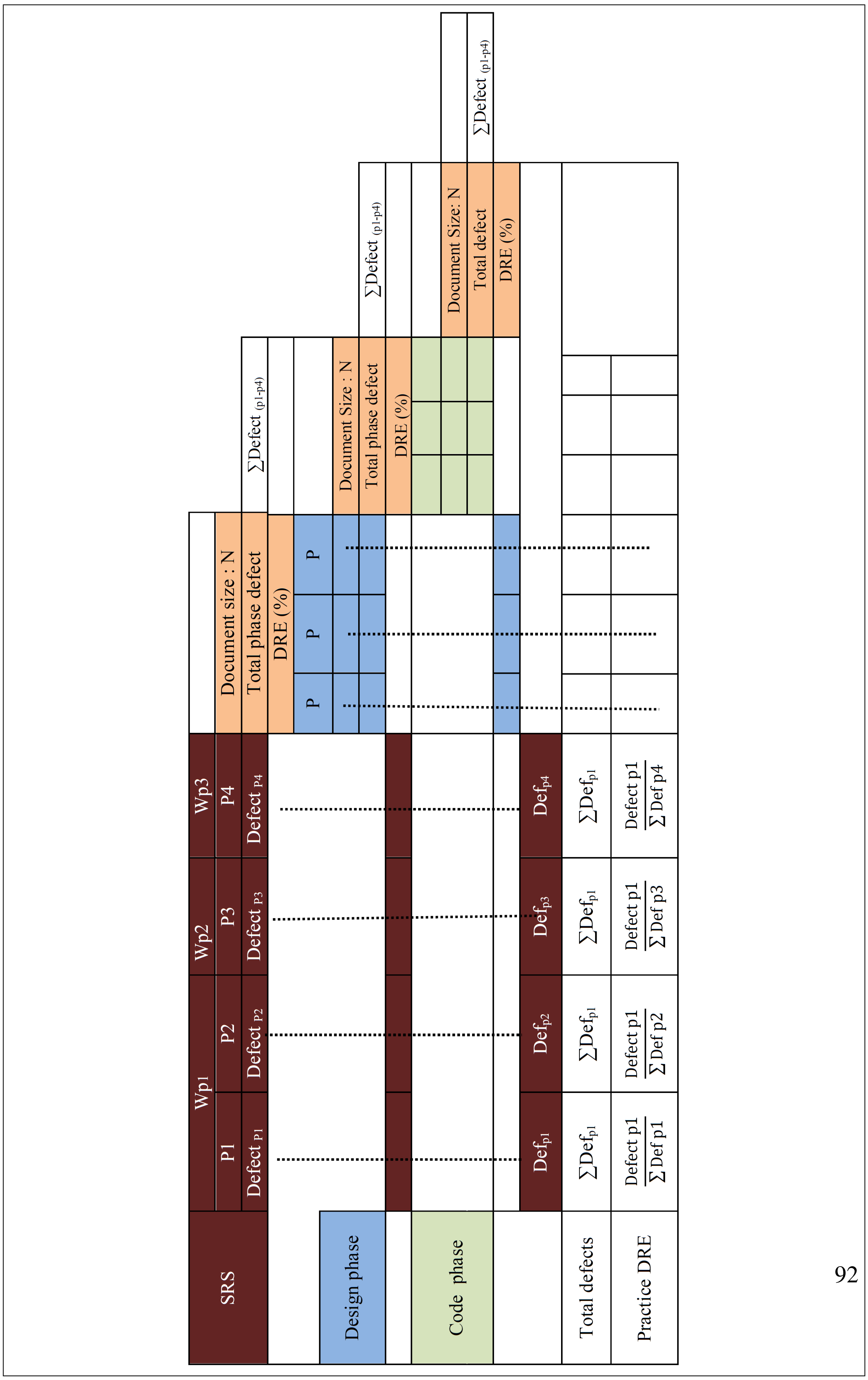


Figure 4.11: Work Product-Based Matrix

In all SDLC phases, each work product will be dealt with differently from others in terms of the QA practice applied to it and in terms of the effort required to run this practice. At the end of the software development process, particularly the system testing phase, the QA team will run the main testing activities like integration testing or unit testing depending on the software's domain of use. As a result of these testing activities, they will uncover more defects which escaped from the main development phases. Each defect discovered will be traced back to its source; that is, which work product this defect originated from, whether is was w_1, w_2 or w_3, and which QA practice, if any existed, this defect should have been found by. As a result of this matching, the *DRE* value for a QA practice with respect to its work product is determined as follows:

$$\boxed{DRE(p_1) = \frac{Defect\ p_1}{\sum Def\ p_1}}$$

- where :

Defect p_1: number of defects found during the work product development process.

$\sum$ *Def* p_1: total number of defects found in the testing phase that belongs to the work product and its associated QA practice.

As can be seen, this matrix shows how to apply the risk-based approach presented in this research. It can store data for each work product categorised and assigned a specific risk level, and it also helps to build a more accurate and decomposed data repository for our QA practices. The demonstration of this matrix's function covered the requirements phase deliverables only which is the SRS; however, it can be extended further to include the design and coding phases following the same approach.

The rationale behind the design of our model is to make it more holistic and more elaborate to help project managers and QA practitioners monitor the flow of data of their

QA activities to the most decomposed level. Therefore, this matrix will help the project manager to determine precisely the *DRE* for each QA practice with respect to both the phase and work products it is applied to. Instead of a vague *DRE* value given by the old defect containment matrix, our matrix is more accurate as it links the removal efficiency to the QA practice used rather than the entire process, and also it quantifies the impact of the work product in terms of its significance and risk with respect to all artifacts of the phase. This advanced defect containment matrix model should be followed within the software development organisation for every software project in order to build up a detailed repository of all QA activities applied and the work products they were assigned to. QA data resulting from following this matrix will go through an analytical process before it is channelled and stored in the repository, as will be thoroughly discussed in the following section.

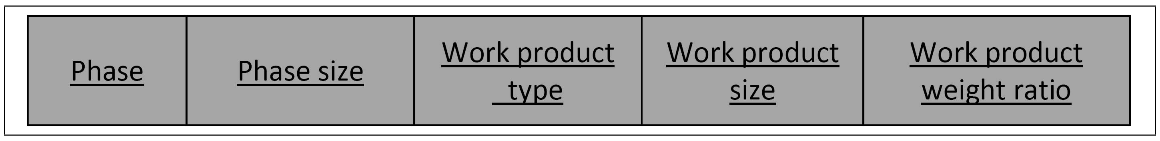
4.7 Data Collection and Analysis

This section shows how data resulting from our model can be sorted and processed in a database so that it can be utilised in making decisions for future usage. As early mentioned, our model helps the software development organisation to make clever decisions concerning their software QA plans by relying on data of past software project developments the organisation built before.

In other words, as in most of the algorithmic software cost estimation models which are based on the analysis of historical data, our model relies on the historical data of past projects but within a single organisation. It stores and analyses data resulting from the software QA activities within the organisation in a way to make it informative and so that it can be used to make future decisions. The main advantage of our model over a similar quality model is that the data used is derived from the software organisation which uses the model itself, the thing that gives more accuracy and credibility to the model. In Section 3.6.4, it was mentioned that the well-known cost estimation model COCOMO is based on the manipulation of 161 previous software projects which may increase the chance of outliers and deviations in its results, taking into consideration the variations in internal process models, personnel, infrastructure, etc., involved with the 161 software projects.

4.7.1 Repository Structure

After the work products categorisation process, the QA team would store all relevant data related to that work product like its size out of the total phase size and its type in the data repository. An overview of the input is depicted below in Figure 4.12.

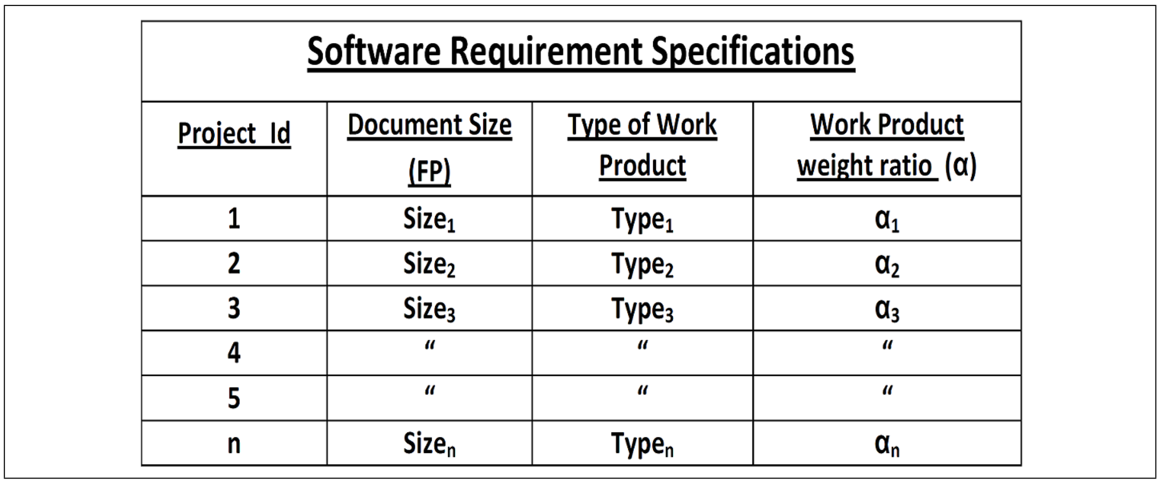


Phase	Phase size	Work product type	Work product size	Work product weight ratio

Figure 4.12: Work Product Details

Following the same approach for several projects, the data of the work products will be sorted and grouped together for each single phase of the software development life cycle (SDLC). In that respect, for every project the following details are stored: the size of every development phase, the number of work products categorised and the weight value of each work product out of the total phase size. An overview of this data entry process associated with a number of projects is shown in Figure 4.13.

Moreover, within each software project and within each phase, data input will decompose further to include two nested tables that include the following:



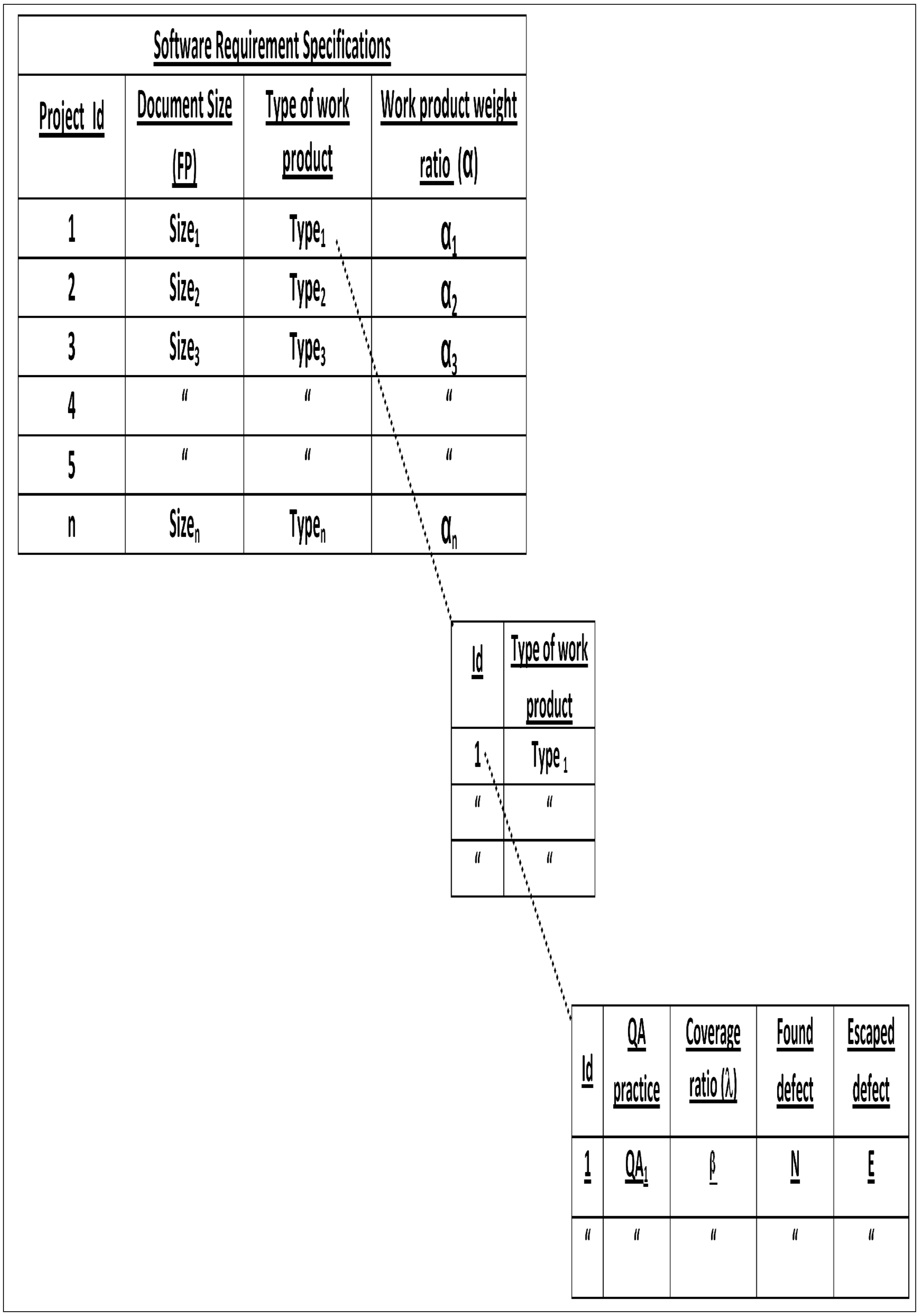
Software Requirement Specifications

Project Id	Document Size (FP)	Type of Work Product	Work Product weight ratio (α)
1	$Size_1$	$Type_1$	α_1
2	$Size_2$	$Type_2$	α_2
3	$Size_3$	$Type_3$	α_3
4	"	"	"
5	"	"	"
n	$Size_n$	$Type_n$	α_n

Figure 4.13: Data Input of Several Projects

- The types of work products for each phase
- QA practices assigned to each work product, coverage ratios given to each practice applied, number of defects found and number of defects escaped using each QA practice.

An overview of the nested input tables is shown in Figure 4.14.



Software Requirement Specifications			
Project Id	Document Size (FP)	Type of work product	Work product weight ratio (α)
1	$Size_1$	$Type_1$	α_1
2	$Size_2$	$Type_2$	α_2
3	$Size_3$	$Type_3$	α_3
4	"	"	"
5	"	"	"
n	$Size_n$	$Type_n$	α_n

Id	Type of work product
1	$Type_1$
"	"
"	"

Id	QA practice	Coverage ratio (λ)	Found defect	Escaped defect
1	QA_1	β	N	E
"	"	"	"	"

Figure 4.14: Input Tables Structure

4.7.2 Data Collection Sheets

In order to simplify the data collection process, two tables that conform to the repository structure described earlier are proposed. Those two tables were designed to utilise the work product categorisation approach of each phase so as to be retrieved easily by our system.

Work-product ID	Type	Size	β (%)	QA Practice	No Defects Found	No Defects Escaped	Cost Detection	Time Detection	Input
0.0.0	RH	10	100	FI	10	5	20	80	{ }
0.0.1	RM	50	100	BC	8	10	2	20	{ }
0.0.2	RL	70	40	BC	3	5	1	10	{ }
			60	none	–	10	–	–	
			100		3	15	1	10	

Table 4.1: Data Collection Sheet

The data collection sheet depicted in Table 4.1 associates attributes with each work product of a phase in a project. We will refer to a specific work product using *project.phase.workproduct* as a unique identifier. Primarily this is the category of the work product. For example work product *0.0.0→Type* is *RH* (Requirement: High Risk); for work product *0.0.1→Type* is *RM* (Requirement: Medium Risk); and for work product *0.0.2→Type* is *RL* (Requirement: Low Risk). For requirement specifications the size of the work product is recorded in *functional points of requirements*.

0.0.0→ β = 100% denotes that all requirements in this work product have been verified using *0.0.0→QA Practice* = FI (Formal Inspection). The activity discovered that *0.0.0→No Defects Found* = 10 defects. Until now *0.0.0→No Defects Escaped* = 5 defects had originated from this work product, but had not been detected by this QA activity. The cost of the detection was *0.0.0→Cost Detection* = £20 and the impact on the schedule was *0.0.0→Time Detection* = 80 hours.

For work product *0.0.2* the data collection sheet shows that two practices have been applied. 40% of the work product was verified using Buddy Checks (BC), whereas the remaining 60% was not verified.

To be able to track the cost and time of defect removal, a more detailed collection of defects must be undertaken. Table 4.2 shows data collected with respect to the defect removal activities. This data is partly available in the amended defect containment matrix.

Defect ID	Defect Type	Cost Removal	Time Removal	Origin Defect ID	Escaped	QA Practice
0.0.0.0	Cr	0.5	0.25	0.0.0.-	no	FI
0.2.1.0	Me	18	10	0.1.1.0	no	BC
0.2.1.1	Me	12	8	0.1.1.0	no	BC
0.1.1.0	Me	2	1	0.0.0.1	yes	BC
0.0.0.1	Me	0.5	0.2	0.0.0.-	yes	FI

Table 4.2: Defect Data Collection Sheet

The first row in Table 4.2 shows that the first defect (*0.0.0.0*) detected (Escaped = *no*) during the Formal Inspection (FI) of work product *0.0.0* is a critical (Cr) defect. Its removal cost was 0.5 k$, and added 0.25 days to the schedule. It has no defect from which it originates (*0.0.0.-*). The second row shows a medium (Me) defect *(0.2.1.0)* that was detected in phase 2, work product 1 of the same project. The removal cost was £18 and added 10 days to the schedule. The defect was traced back to *(0.1.1.0)* that escaped from phase 1. The defect was found using Buddy Check (BC). The defect in the third row is similar to the second and also traces back to the same origin *(0.1.1.0)*. The fourth row of Table 4.2 shows a defect *(0.1.1.0)* which is a Medium (Me) defect that escaped (Escaped = *yes*) the Buddy Check (BC) verifying work product 0.1.1. It could be traced back to its origin (0.0.0.1, row five), a defect from the requirements phase's work product 0.0.0. This defect was not detected by the formal inspection verifying these High Risk Requirements (see Table 4.1) and cannot be traced back any further.

4.8 Data Analysis Process

Having completed the data input process, the data analysis process starts by grouping the dependent variables together to analyse and determine the relationships between them (Table 4.3). The mechanism on which our analysis process is based is the average values of variables and the regression analysis.

Software Requirement Specification			
Project$_{id}$	Document size (FP)	Type of work product	Work product weight
1	Size$_1$	Type$_1$	$\alpha_1\%$
1	Size$_1$	Type$_2$	$\alpha_2\%$
1	Size$_1$	Type$_3$	$\alpha_3\%$
...	....	....	{100%}
2	Size$_2$	Type$_2$	$\alpha_2\%$
2	Size$_2$	Type$_3$	$\alpha_3\%$
3	Size$_3$	Type$_3$	$\alpha_3\%$
4	Size$_4$	Type$_4$	$\alpha_4\%$
5	Size$_5$	Type$_5$	$\alpha_5\%$
.	.	.	.
.	.	.	.
.	.	.	.
n	Size$_n$	Type$_n$	$\alpha_n\%$

Table 4.3: Process Input Table

Our average values determination process will analyse all QA data of past projects that are stored according to our model structure. This process will be applied to the following variables and database tables:

1. Each QA practice with respect to the phase and work product type (Table 4.4).

QA practice : P$_1$					
Project$_{id}$	α	**Coverage weight**	**Defects found (FD)**	**Defects escaped (DE)**	**DRE**
1	α_1	β_1	**FD$_1$**	**DE$_1$**	**DRE$_1$**
2	α_2	β_2	**FD$_2$**	**DE$_2$**	**DRE$_2$**
3	α_3	β_3	**FD$_3$**	**DE$_3$**	**DRE$_3$**
4	α_4	β_4	**FD$_4$**	**DE$_4$**	**DRE$_4$**
n	α_n	β_n	**FD$_n$**	**DE$_n$**	**DRE$_n$**

Table 4.4: Defects found and removed by QA practice P$_1$

The result of this table will determine the average *DRE* value for a specific QA practice with respect to the work product and to the phase it is applied to.

Phase	Work product	QA practice	$D\bar{R}E$

As discussed before in Section 2.1.2, QA practices are diverse in their applicable domain and vary significantly in their efficiency of defect removal and in their effort for execution and defect removal. As this research proposes a new approach that relies entirely on categorised work products, using average data regarding the defect removal efficiency taken from literature or industry reports is not feasible and not accurate. Therefore, the DRE value for each QA practice applied to each work product needs to be determined from the advanced defect containment matrix proposed in Figure 4.11.

2. The execution time for each QA practice with respect to the phase and work product type (Table 4.5).

QA practice : P$_1$ type$_1$			
Project$_{id}$	**Coverage weight**	**Size**	**Execution time**
1	β_1	β.**Size**$_1$	**time**$_1$
2	β_2	β.**Size**$_2$	**time**$_2$
3	β_3	β.**Size**$_3$	**time**$_3$
4	β_4	β.**Size**$_4$	**time**$_4$
n	β_n	β.**Size**$_n$	**time**$_n$

Table 4.5: Execution Time for QA Practice P$_1$

The result of this analysis process is to determine the average time ($\bar{time}$) value needed by QA practice p_1 to run on a work product w_x.

Phase	Work product	QA practice	$\bar{time}$

3. Each QA defect removal cost with respect to the phase and work product type Table 4.6.

QA practice : P$_1$: type$_1$		
Project$_{id}$	**No. of found defects**	**removal cost**
1	**defect**$_1$	**cost**$_1$
2	**defect**$_2$	**cost**$_2$
3	**defect**$_3$	**cost**$_3$
4	**defect**$_4$	**cost**$_4$
n	**defect**$_n$	**cost**$_n$

Table 4.6: Defect Removal Cost For a Single QA Practice

Similar to *DRE*, the effort needed for each QA practice to execute the artifact inspection process to be quantified. It is hard to quantify the relationship between the value of *DRE* and effort. Instead, industry would take average values of effort related to any QA practice. For this reason, our model uses the average cost value related to removal of a defect using a QA practice. The result of this association will be:

$$\boxed{\text{Phase} \quad \text{Work product} \quad \text{QA practice} \quad \bar{cost}}$$

4.9 Size and Total Defects Relationship

In order to build up the decision support system that will be based on our model variables, the relationship that links the number of defects injected in every work product with its size needs to be identified. This association will be based on the work product type and the software development phase it belongs to (Table 4.7).

Software Requirement Specification		
Project$_{id}$	Work product size	Total defect of the work product
1	type$_1$	total-defect-type$_1$
2	type$_1$	total-defect-type$_1$
3	type$_1$	total-defect-type$_1$
4	type$_1$	total-defect-type$_1$
n	type$_1$	total-defect-type$_1$

Table 4.7: Work Product and Total Defects

As in Table 4.7, QA data of five completed projects was retrieved which shows the number of defects found in every work product of a specific type, *type*$_1 \in phase_x$, and the size of that work product. Any relationship that exists between the work product size and the total number of defects injected needs to be uncovered. Also, the variations within these relationships during the different past projects needs to monitored and recorded. As a result of an extensive literature research of software defects data and software quality models, it is found that the number of defects injected into a piece of software correlates with the size of the software itself in different forms. The following text shows the different points of view with respect to this relationship and the supporting evidence for each.

With the ongoing increase in software size and its development activities, there have been many research studies that have tried to show the interrelationships between software variables. One of these variables was the software defect size and number of defects or, in other words, the software size and its defect density. Generally, those results show that defect density and software size have a strong relationship and that a correlation exists between the two variables. This relationship may take two forms: linear relationship or a non-linear relationship.

- Linear Relationship

  A considerable amount of studies have revealed the linear relationship between software size and its defects. As an example, Akiyama [3] discovered this relationship using data from nine modules programmed using Assembly language. Halstead [60] and Funami [51] found a strong correlation between the size of their software and the number of defects injected and that correlation was linear. Another recent study was conducted by Gimothy [59] who relied on data from NASA projects and found a clear linear relationship between object-oriented software size and its injected defects. Moreover, in a recent study published in 2007 [75], a relationship was found between software size and total number of defects. This studies revealed that with growing software size there is a tendency in to inject more defects. A detailed overview of the literature on the linear relationship between software size and number of defects can be found in [87].

- Non-linear relationship

  On the other hand, some researchers observed a non-linear relationship between the software size and the number of defects injected [10, 87]. This relationship was more logarithmic rather than linear. However, such non-linearity is not accurate and there are significant variations amongst studies with respect to the logarithmic value found [95, 103].

In our approach, particularly where our model feeds on data taken entirely form the software organisation applying the model, the suggestion is that the linear correlation between software size and the delivered defects will occur. The rationale behind that is that the software development process is an integration of several variables and factors; these factors contribute with different magnitudes to the attributes of the final software. Such factors are: software development personnel, the software process model implemented, the type of the software etc., so therefore, by keeping these factors stable, which is the case in our model, the relationship between software size and number of defects is stable. This is also supported by the fact that this relationship is between work products of the same type and that belong to the same phase which is not the case for the previous studies.

However, for more accuracy and in order to get rid of any outliers that may occur within the data collection process, the assumption is that this linear relationship will always occur. Instead, a stage is devise to check the correlation between work product size and its overall defects found as shown in Table 4.7 that precedes the regression analysis process. The correlation check process will be applied for at least five or more software projects developed inside the organisation in order to verify the positive linear relationship between the work product size and the number of defects found before relying on the model as a quality estimation tool.

4.9.1 Pearson Correlation Check

The correlation analysis is a process by which the linear and non-linear relationship between two variables are measured and determined. There are two well-known correlation analysis methods: the Spearman coefficient and the Pearson coefficient for measuring the linear and non-linear correlation coefficients respectively [156]. As this research is built on the fact that there is a linear relationship between the size of the work product and its total number of defects found, the Pearson coefficient is applied to determine the direction and the strength of this relationship.

In the Pearson coefficient, the correlation between two variables (x, y) can take three forms:

- Positive correlation

  A positive correlation means that an increase in the value of **x** is faced with an increase in the values of **y**.

- Negative correlation

  A negative correlation means an increase in the value of **x** is faced with a decrease in the values of **y**.

- Weak correlation

  In this type of correlation, the relationship between the two variables are to some extent independent and cannot be captured.

In our study, **x** denotes software size and **y** represents the number of defects.

In order to apply the Pearson correlation, the following equation is used to get the coefficient value *(r)*:

$$\boxed{\mathbf{r} = \frac{\sum XY - \frac{\sum X \sum Y}{N}}{\sqrt{(\sum X^{2\frac{(\sum x^2)}{N}}) \sum Y^{2\frac{(\sum y^2)}{N}})}} \quad Source : [129]}$$

The value of *r* have a range of between (-1 to 1) and is interpreted as follows:

- +1: Strong positive correlation
- -1: Strong negative correlation
- 0: No correlation

The closer the value of **r** is to **+1** or **-1**, the more strong is the relationship between the two variables **x** and **y** which means that there is indication of a linear relationship between the two variables. As a result, this relationship can be quantified using the linear regression analysis. Conversely, the closer the value of **r** is to zero the weaker the relationship is and should not be considered. An overview giving examples of the **r** value is depicted in Figure 4.15.

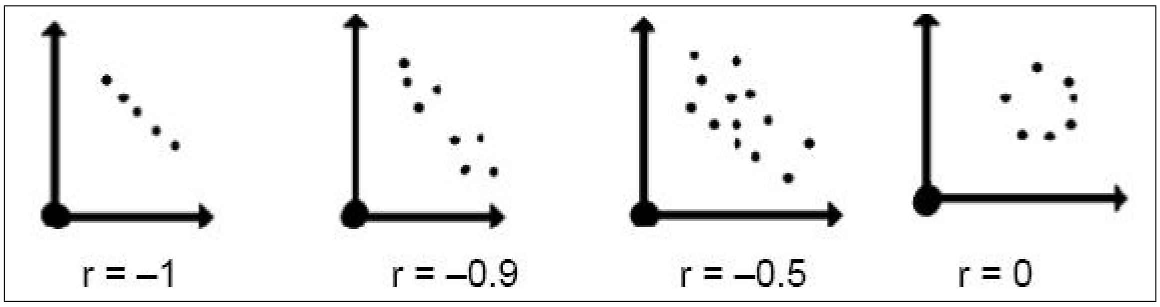


Figure 4.15: Examples of Nositive Pearson Correlation

As discussed before, in our approach, the expected value of *r* which measures the relationship between the number of defects and the software size should be $\geq$ 0.5.

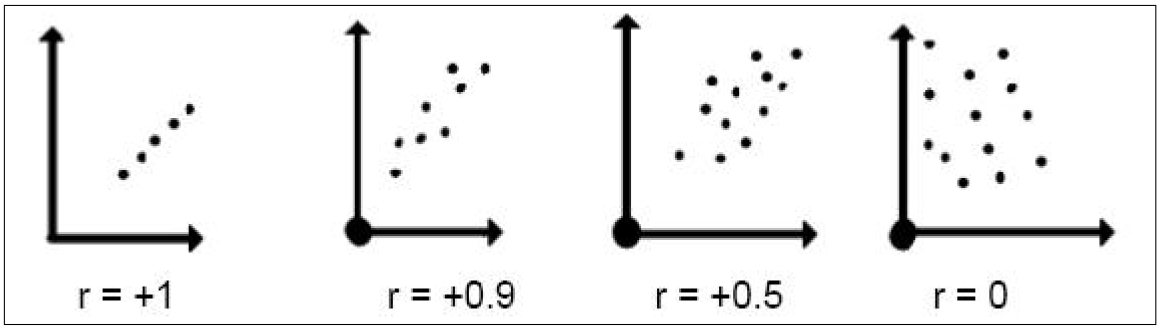


Figure 4.16: Examples of Positive Pearson Correlation

4.9.2 Linear Regression

Once the positive correlation is determined, linear regression analysis is applied to the two variables: work product size denoted by x and the total defects represented by y. An overview of the two variables is shown in Table 4.7. The goal of linear regression analysis for a set of data points is to solve the following equation denoting the best-fit trendline between those data points:

$$\boxed{y = m * x + b}$$

Where:

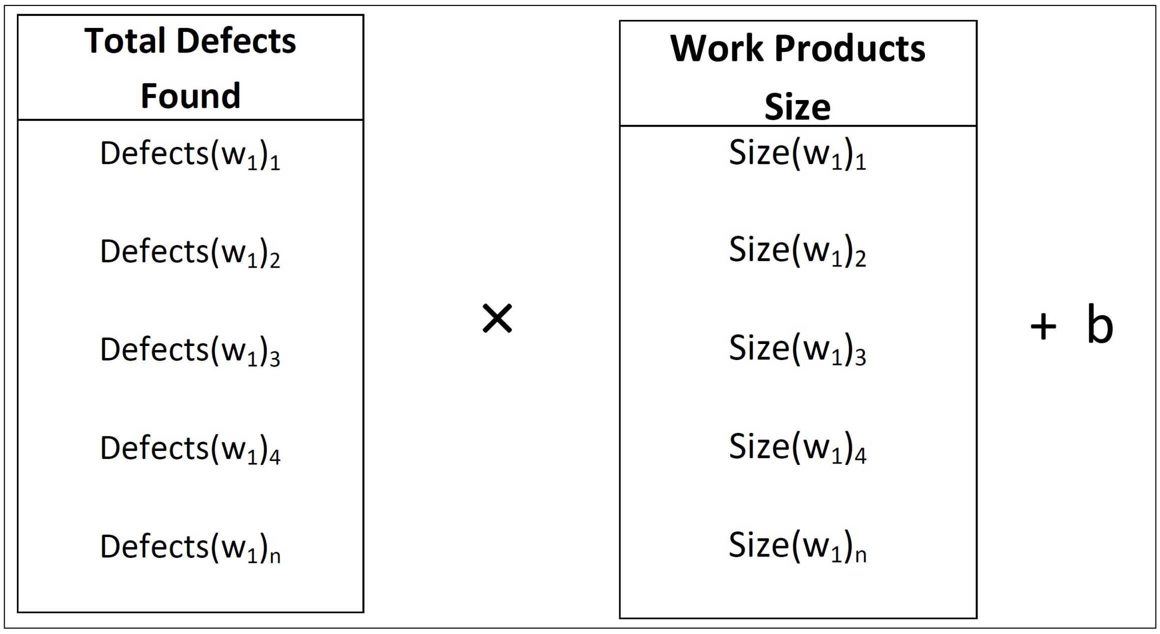
y: number of injected defects in a work product.

x: the size of the software work product.

m: the slope-intercept between the two variables.

b: is a constant.

In order to solve the previous best-fit line equation, Least Square Method is used and applied to the target Table 4.7 of the size of the work products and total associated defects.



4.9.3 Linear Equation Mapping

Having completed the regression analysis process, the equation resulting from the plotting of the values of the two variables on a regression analysis graph (Figure 4.17) links the number of defects with the size of the software work product as follows:

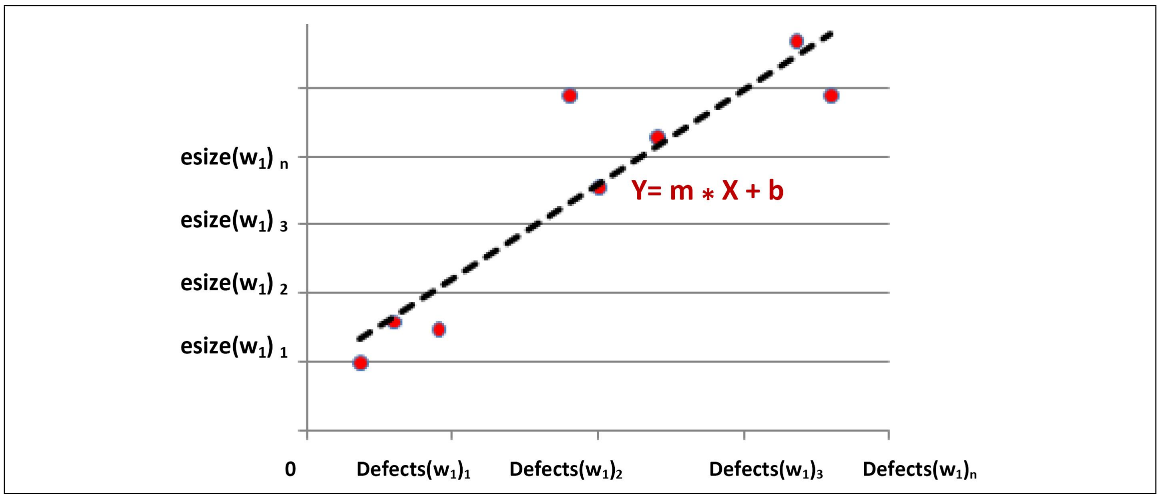


Figure 4.17: Proposed Regression Analysis

- where:

$$Size(w_1)_1 \rightarrow \qquad \rightarrow Size(w_1)_n$$

and

$$Defects(w_1)_1 \rightarrow \quad \rightarrow Defects(w_1)_n$$

- represents the size of a work product of a specific risk type retrieved from several previous projects and its relative total defects found at each project iteration. Accordingly, the QA team can use the equation as a baseline in order to predict the total number of defects estimated to be injected in work product of a specific type into a specific phase of the software development life cycle. The flowchart of the implementation of the model is depicted in Figure 4.18.

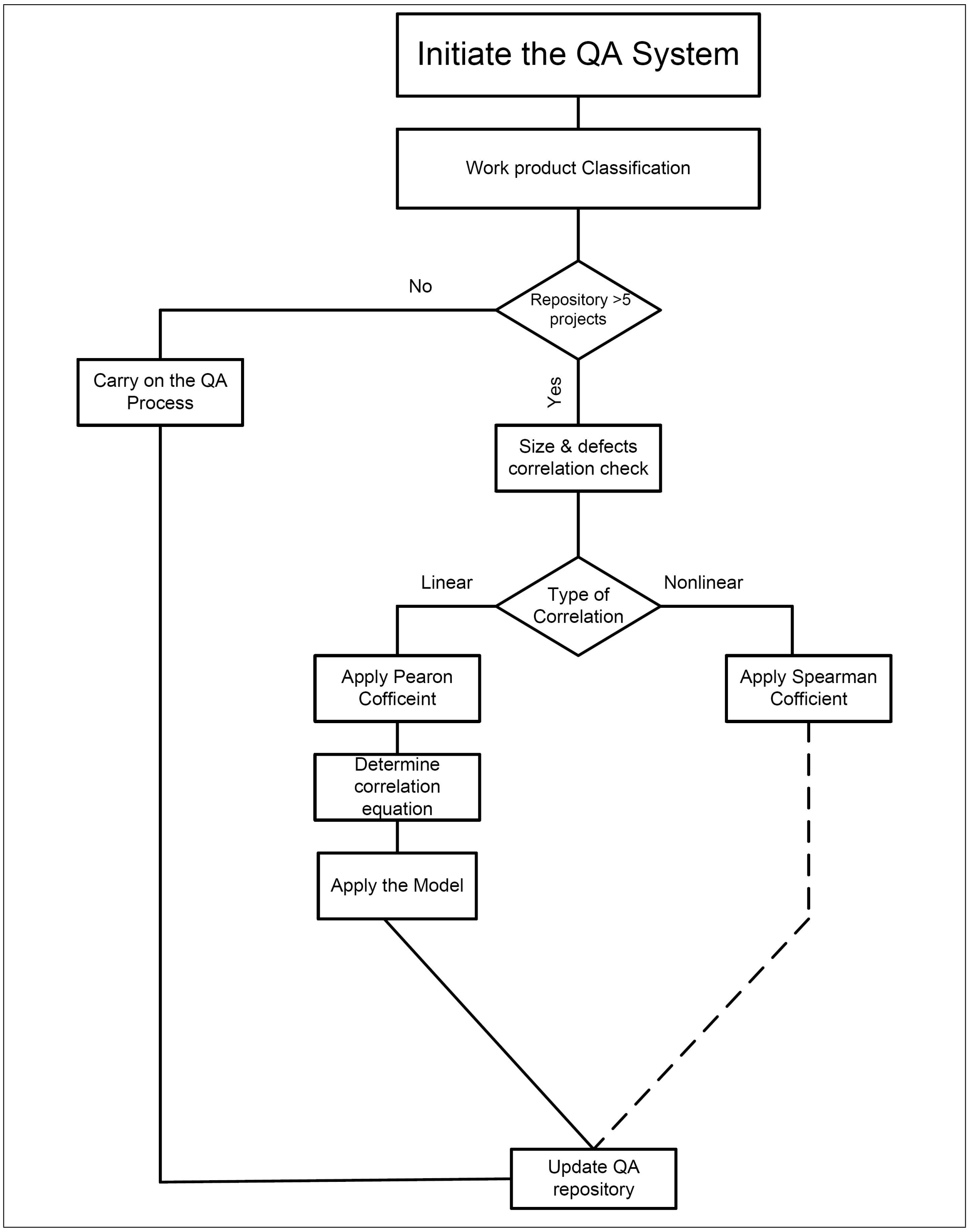


Figure 4.18: Flowchart of Work Product QA Plan

4.10 Time Constraint

Triple constraints, discussed in Section 3.4, need to be efficiently managed and controlled by the software project manager to assure development success. Referring to the triple constraint triangle shown in Figure 3.4 of Section 3.4, the three constraints should work in tandem so as to release a software that meets its pre-defined scope. So, if the time aspect of a software project was reduced, this would impact on the final quality of the software due to the fact that there is less time given to analysis, testing and monitoring of activities.

4.10.1 Work Products & Development Tasks

In our approach, where risk-based quality is the main incentive, classifying work products would help in refuting the current conviction that a reduction of the schedule would negatively impact on the quality of the final software. This section shows the ability of our model to control this issue which would help to make the trade-off within the triple constraints problem. As an example, our work product categorisation process reflects a normal development process as is shown in Figure 4.19.

As can be seen, the rectangle in the middle of the figure represents the development activities within a single phase. D_0 denotes the deliverable coming in from the preceding phases and D_1 denotes the deliverable going to the subsequent phase. Within the development phase, there are four different development tasks referred to as *$Task_1$, $Task_2$, $Task_3$ and $Task_4$*. In that phase, our risk levels are assigned as follows :

$Task_1$ & $Task_3$	Low
$Task_2$	High
$Task_4$	Medium

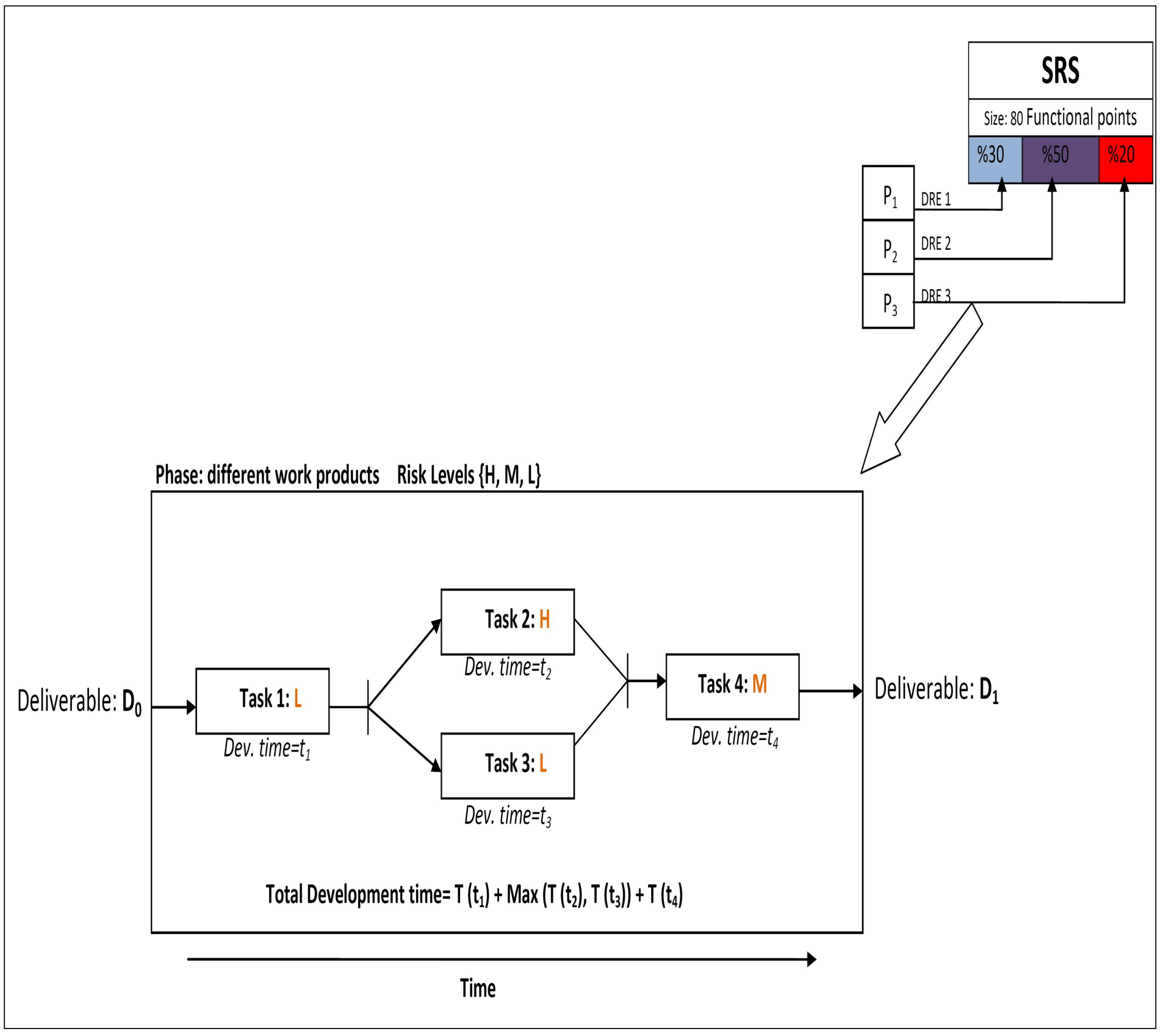


Figure 4.19: Time Constraint of Work Product Development

The development process of the tasks is sequential; that is, each task relies on the previous task's output. However, we assume the scenario that some of the tasks are developed in a parallel way. For example, as shown in Figure 4.19, Task$_2$ and Task$_3$ rely on the input of Task$_1$. Accordingly, both tasks can be developed concurrently.

For each task, there is a development time denoted by t_1, t_2, t_3 *and* t_4, which added together is the total phase development time represented by the long arrow down the figure. The overall phase time needed is represented as a function of the sum of all tasks development times as follows:

$$\text{Total development time} = T(t_1) + Max(T(t_2), T(t_3)) + T(t_4)$$

So, the times of *Task*$_1$, *Task*$_4$ are added with the max value of Task$_2$ and Task$_3$ due to the fact that their development activities will be carried out concurrently.

4.10.2 Applying QA Practices

For the next step, the project manager will assign the required QA practices to each task to improve the quality and to reduce the defect injection rate.

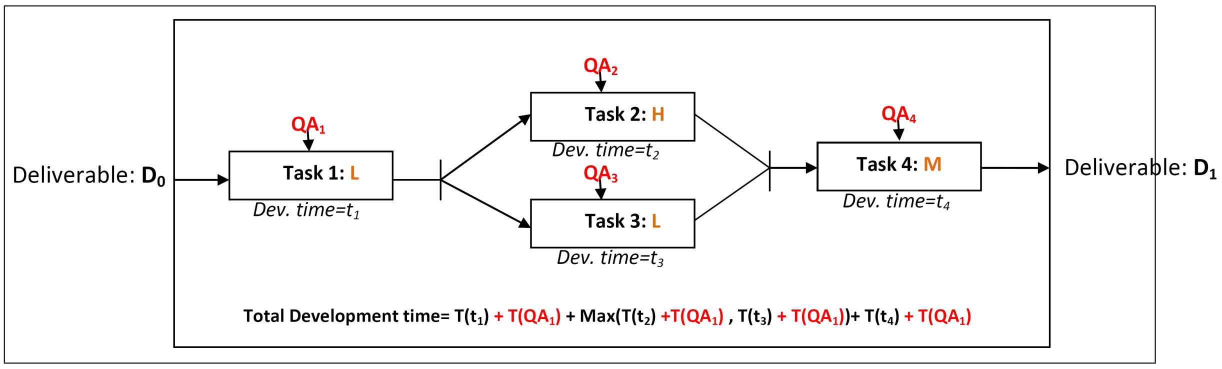


Figure 4.20: Time Constraint of Work Product Development

In Figure 4.20, the QA practices are assigned to each task as follows:

Task_1	QA_1
Task_2	QA_2
Task_3	QA_3
Task_4	QA_4

In this case, the total development time of the phase will include the execution and removal activities for each QA practice as follows:

$$\textit{Total development time} = T(t_1) + T(QA_1) + Max\ (T(t_2) + T(QA_2)\ ,\ T(t_3) + T(QA_3)\ ) + T(t_4) + T(QA_4)$$

4.10.3 Time Reduction

Now, let us suppose that in using a cost estimation tool like COCOMO II, the estimated development times for the four tasks are *2M (month), 3M, 4M and 2M* respectively (Figure 4.21). In this case, the total development time for the whole phase excluding the QA practices applied is 8 months.

Due to some constraints, the project manager decided to cancel the investment given to QA processes for the task of work product type *L*. This means that QA practices applied to Task_1 and $Task_3$ are cancelled. By looking at the concurrent tasks $Task_2$, $Task_3$, the minimum total development time is four months which means that $Task_2$ will be idle for one month. Accordingly, the project manager may use this time to inspect $Task_2$ which represents high criticality to the phase without the need to delay the whole phase and hence the whole schedule.

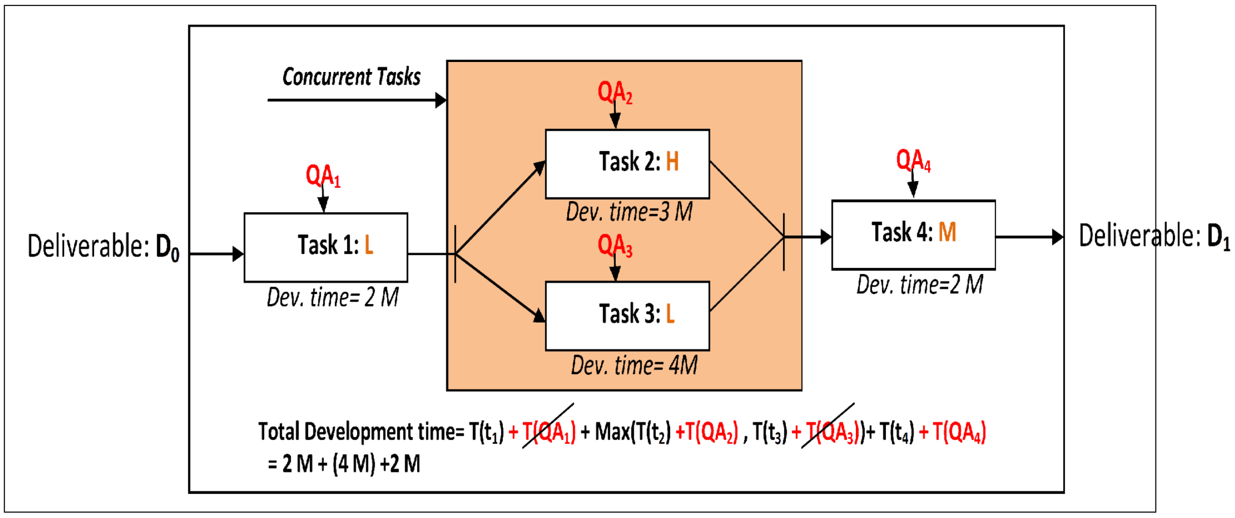


Figure 4.21: Time Reduction of Development Tasks

This scenario sheds a light on how our approach of risk-based QA may help in reshaping and reconstructing the relationship between **Quality** and **Time** as linearly related dimensions. In other words $\uparrow$ **Quality** $\longrightarrow$ $\uparrow$**Time** which represent the classic unwise QA plan can be replaced by $\uparrow$ **Quality** $\longrightarrow$ **Time**. Moreover, if it is taken into account how defects escaping from the high risk $Task_2$ can affect the delay of developing $Task_4$ it can be suggested that the relationship between **Quality** and **Time** can change to $\uparrow$ **Quality** $\longrightarrow$ $\downarrow$ **Time**.

4.11 Summary

The application of this approach to many projects following the same software development process enables the QA team to build up a data repository of their QA activities. This data repository will include defect reports that contain the number of defects solved for each work product and the number of escaped defects from applying each QA practice. This chapter outlined the data collection and analysis processes of our model utilising the proposed defect containment matrix. The process of checking the linear relationship between the work product size and the number of defects is defined. This chapter also outlined the categorisation scheme of work products based on pre-defined risk levels. Also, a solution to the time constraint problem was proposed which utilises our risk-based categorisation approach of software development phase. The following chapter presents the formal expressions of our model to link its components together and to help generate optimal decision for QA activities.

Chapter 5

Formal Model

Objectives

- To present the theoretical model formally.
- To determine the components of the model and how they interrelate with each other.
- To measure the effect of choosing a specific QA technique over another.
- To facilitate the process of counting the number of defects detected and removed by a QA practice.
- To calculate the cost of execution of using a QA practice.
- To calculate the execution duration needed to run a QA practice.
- To re-define the return on investment (ROI) metric and how it can be integrated within the model.

5.1 Introduction

The previous chapter presented our model theoretically by defining its main components and its main internal processes. It showed how the data collection and analysis process can be carried out and manipulated to determine the values needed for our model variables. Having completed those steps and with the model repository having a considerable amount of QA data, the system can then be used to generate informative decisions of QA activities.

This chapter identifies the formal model of our system which forms the basis of the decision making process of QA activities. The formal model consists of three interrelated components: **1- the estimated number of found and escaped defects**, **2- the execution cost and time of a QA practice** and **3- the escalation cost of escaped software defects**. Formal equations are presented for each component that utilises the output of the data collection and analysis process. Then based on these equations, decisions on QA activities can be formed and generated.

The proposed formal expressions rely on the linear relationship approach between software size and its defects that are followed in this research. However, they are presented in a way to be easily customised to include non-linear or non-parametric relationships. The proposed formal model also conforms to our model's framework presented earlier in Section 4.1 to help both models work as a unified structure.

5.2 Formal Model Structure

5.2.1 Development Phases Structure

As discussed before, our model is applicable to any software life cycle model as long as it consists of separate phases such as ***Waterfall***, ***Rapid Application Development*** etc. Most software development life cycle models divide the development process into a number of phases or stages. Each phase is in turn divided into steps or sub-processes. A typical life cycle model is the waterfall SDLC discussed in Section 2.3. More modern approaches to software development use iterative or agile methodologies in which the software system is not produced in such a serial manner, but as an incremental, repetitive sequence of smaller implementation tasks. Our approach is not dependent on any particular life cycle model, but assumes that the model has clearly identifiable phases, that take higher level work products (e.g. a design document) as an input and create lower level work products (e.g. code or test suites) as an output.

Following the SDLC approach being used by the software development organisation, our software quality model consists of a sequence of phases: requirements, design and code which are referred to as (r, d, c). An overview of the phases structure is depicted in Figure 5.1.

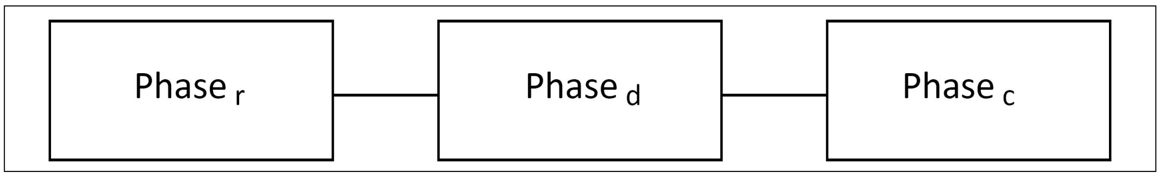


Figure 5.1: SDLC Phases

5.2.2 Work Products Structure

The next step after defining the software phases is to define the work products within each phase. As discussed before in Section 4.10, the activities within a single phase can be multi-tasked and divided into sets of work products depending on the risk consideration given by the project manager to each phase's work products. Therefore, each phase consists of a number of work products $w_1 \rightarrow w_n$. The number of work products for each phase is dependent on the risk consideration levels determined and agreed upon by the software project manager and the QA practitioners. Note that these levels are fixed per phase during the application of the system. An overview of the work products and phases association is shown in Figure 5.2.

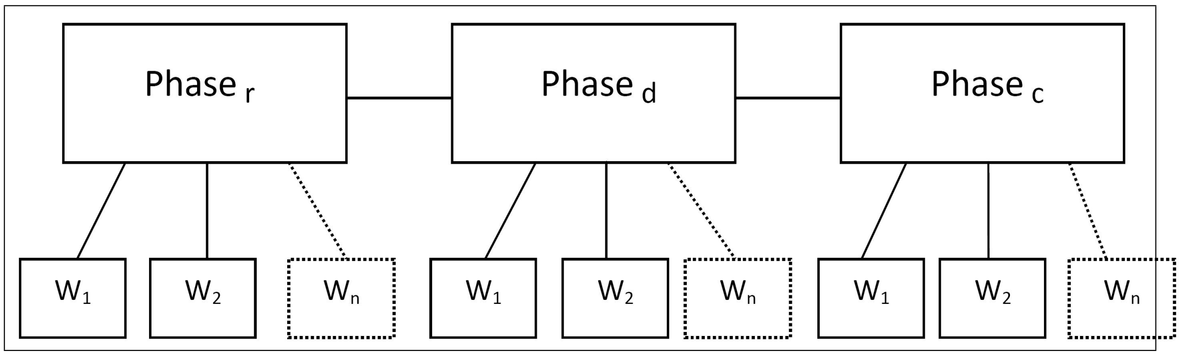


Figure 5.2: Work Products

5.2.3 QA Practices and Work Products Association

Having defined the software life cycle phases and their work products, QA practices available within the organisation need to be grouped and classified. There are a number of QA practices which are specifically used and responsible for the defect detection and verification activities within each phase. These QA practices are assigned a domain of applicability within the software life cycle so that each QA practice cannot be used anywhere.

Referring back to Section 2.5.2, it were shown that there is a diversity in the defect detection and removal techniques used currently by the QA practitioners in the software development industry. This diversity introduces some key variations between those techniques in terms of the followings :

- The execution time of the QA practice.
- The execution cost of the QA practice.
- The applicable domain and relevance of the QA practice to a software artifact and to a software development phase.

Therefore, classifying QA practices based on a per phase basis helps in identifying the most suitable and what best fits the QA activity within that phase. By completing phases and the work product categorisation process and by grouping QA practices for each phase, the overall classification structure can be comprehensively depicted as is shown in Figure 5.3:

Based on that model, for each $phase_x$ where $x \in r, d, c$, there is a number of work products $(w_1 \rightarrow w_n)$ and a list of QA practices $(p_1 \rightarrow p_n)$.

5.2.4 Quantification Cost of QA Activities

The role of each QA practice within each work product of the software life cycle phase includes the defect detection and removal activities. These activities will then typically take place as part of the phase to verify that the outputs match the conditions imposed by the inputs on the phase. Each QA activity within the phase is allocated time and effort as part of the overall development plan. As shown in Figure 5.4, a software life cycle phase

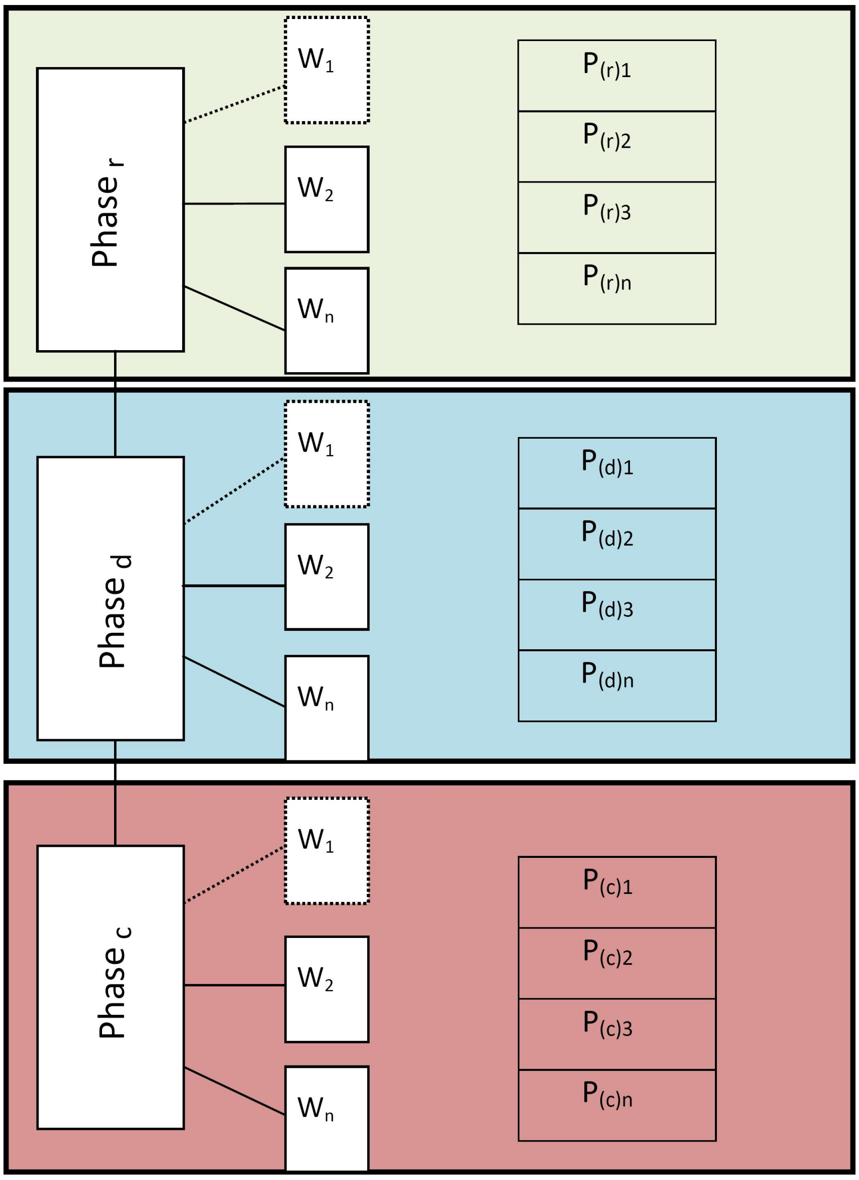


Figure 5.3: Comprehensive Model

takes input from the preceding phase and delivers output to the succeeding phase. In between the input and output, the QA activities carried out within the phases include the actual development activities and the verification activities for QA purposes. The result of these activities within each work product, not the whole phase, is what our model tries to quantify and calculate. Accordingly, the allocated time and effort for the QA activities is going to be optimised as a result of this quantification, which will reduce any unnecessary investment given to such activities.

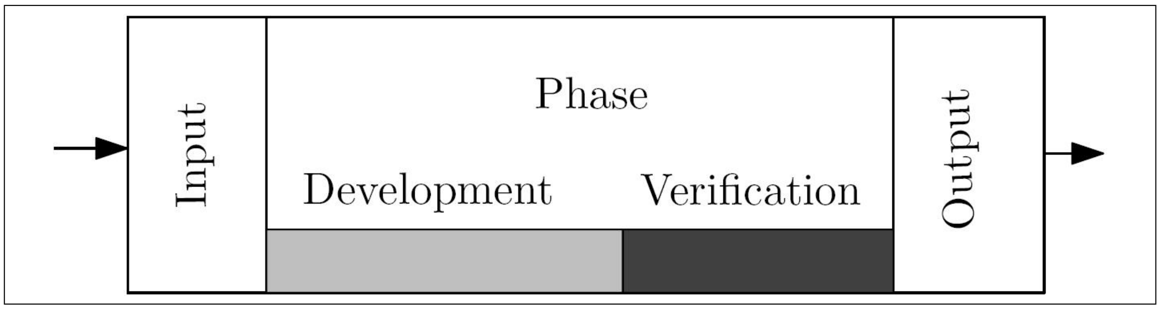


Figure 5.4: Internal Activities of the Phase

5.2.5 The Model Components

In the beginning of this chapter, it was stated that the main role of the formal model is to control the data collection of our regression model and to utilises values resulting from QA activities such as, *DRE*, $\bar{cost}$ and $\bar{time}$. These values will be used by the model in order to make QA decisions of potential and future defect removal and detection activities. Therefore, each value resulting from the data collection and analysis process need to be looked at to determine the other dependent variables that affect its final value.

For example, for any QA activity the main variable associated with the *DRE* value is the number of defects found to the number of defects escaped (Section 2.6.1). This also applies to the other values of $\bar{cost}$ and $\bar{time}$. So for that reason, our formal model set out formal equations to calculate each value as a function of its related variables.

Generally, the formal model of our system consists of three main components as follows:

1. **Number of Defects (I)**

   This component refers to the estimated number of defects found by applying a specific QA practice to a single work product (w) or to a whole phase in general. The component of the number of defects of our model will be responsible for determining the DRE value with respect to the QA practice used.

2. **Execution Cost and Effort**

   The execution cost is the cost of performing the QA process using a specific QA practice. As the aim in this research is to optimise the cost of QA investment, this value needs to be accurately quantified in order to reduce the waste in effort introduced by any QA activity. In our model, the cost of execution is divided into two blocks:

   - Cost to run the QA practice.
   - Cost to remove defects discovered.

   The reason for this division is that during a QA activity, which comprises defect detection and removal, the defects found or some of them may not get removed during the QA activity even though they were discovered. The QA team may prefer to remove only part of the defects that were found due to unexpected constraints or because of the fact that it is not necessary to fix all of the defects. For example, as shown in Figure 5.5, a QA practice p_1 was applied to an untested software artifact to determine how many defects were injected before starting the defect removal process. This process introduced an execution cost, represented by the time arrow down the figure, despite the fact that no defects were removed. The discovered defects then go through a removal process which in turn introduces a removal cost.

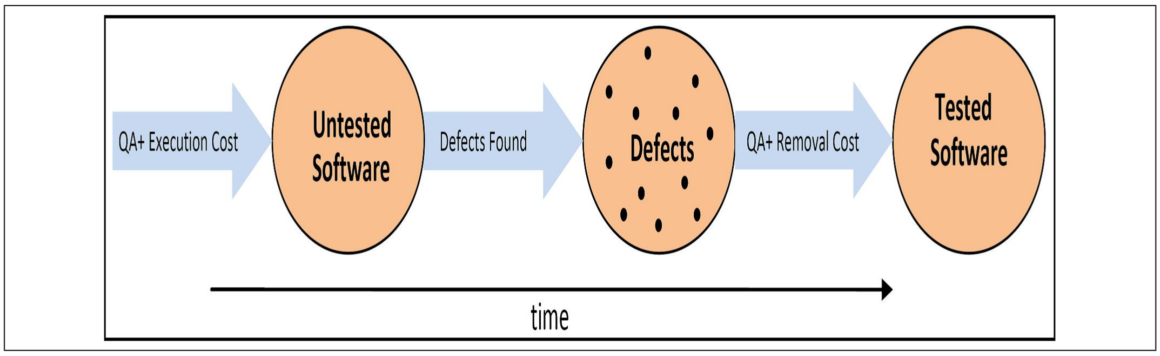


Figure 5.5: Execution and Removal Cost

For that reason, this research distinguishes between the cost of execution and the actual defect removal cost, which are referred to in our regression model as $time$ and $cost$ respectively. See Section4.8. The main output of this component is to define the $c\bar{o}st$ and $ti\bar{m}e$ values.

3. **Cost of Escaped Defects**

   Referring to the QA example shown in Figure 5.5, some defects which were overlooked by the QA team will certainly propagate to the later phases. These escaped defects need to be quantified in terms of the estimated cost associated with their defect removal activities. Doing so will help in covering all aspects of cost related to a specific QA activity.

   This component of our model quantifies the impact of defects escaped from the main development phases of the SDLC: requirements, design, coding, etc., with respect to the system testing stage. An overview of this association is depicted in

Figure 5.6.

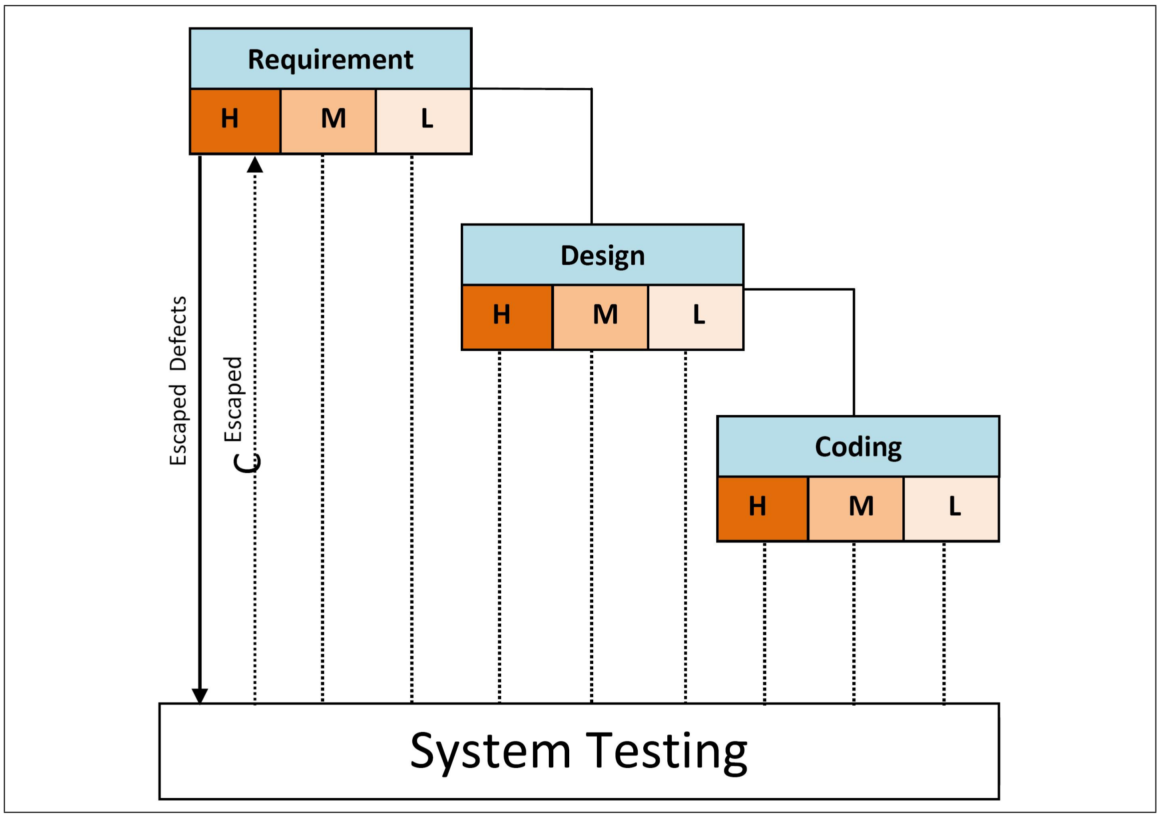


Figure 5.6: Cost of Escaped Defects

As discussed earlier in Section 4.2.1, the mechanism of our model works by associating QA activities carried out during the development activities of the SDLC's phases with the system testing phase, where the main testing activities begin. In the system testing phase, defect detection and removal techniques uncover defects resulting from the development activities and source them back to their origins using the advanced defect containment matrix proposed in Section 4.6.

In our model, this sourcing mechanism should quantify the impact of all escaped defects in terms of their removal cost and time with respect to the work products they belong to in the software development phases.

As a result of this process, it will be possible to measure the estimated cost introduced by a defect that escaped from a work product of a specific risk level and was discovered in the system testing phase. This value is referred to in our model as the escalation factor of a defect ($C^{escaped}$). As will be shown later, the escalation factor will be utilised by our model for the trade-off process of QA alternatives and to generate optimal solutions to QA plans.

5.2.6 Model Variables & Notations

This subsection describes the variables used in our model and their definitions based on our model structure shown in Figures 5.1, 5.2 and 5.3.
These variables will be used by our model as an input to make the required calculations.

- There is x number of software development phases ($phase_x$) where $x \in r, d, c$ representing the requirements, design and coding phases respectively.
- For each $Phase_x$, there are a number of work products (W_x) categorised according to pre-determined risk levels or types where w corresponds to risk rating {High, Medium, Low}. See Section 4.4.
- P_x denotes the set of all QA practices that can be applied in $Phase_X$.
  Let $p \in P_x$
  Let $w \in W_x$
- C$_p^{removal}$ refers to the cost of removing a defect using a QA practice p. This cost is measured in a unit of time per defect scale. Note that the value of C$_p^{removal}$ is relative to a specific work product's type within a specific software development phase and removed by a specific QA practice. For more details see Section 4.8.

- DRE_p is the defect removal efficiency value for a QA practice p. Again it should be noted that the DRE value of the QA practice p is relative to the phase and the work product type it is applied to. A QA practice p can have two different DRE values with respect to different work product categories or within the same work product category but in a different software development phase. For more details see Section 4.8.

- C^{escaped} refers to the escalation factor of an escaped defect with respect to the system testing phase. Note that C^{escaped} is associated to a specific work product type and a specific development phase of the software life cycle. In this variable in particular the effect of the QA practice is excluded as defects are removed using testing techniques within the system testing phase.

- β_p refers to the coverage weight of a practice p during a QA activity. When applying a QA practice to a work product $w \in W_x$, the QA team should use β to determine the testing coverage of this practice out of the whole work product's size. For example, when applying n (number of practices) to a single work product the assignment process will be as follows:

$$\beta_{p1} + \beta_{p2} + ... + \beta_{pn}$$

It is essential to know that the sum of beta values for a single work product should not exceed the actual size of the work product. This coverage condition is expressed as:

$$\sum_{p \in P} \beta(p) = 100\% \text{ of coverage ratio}$$

5.3 Number of Defects Found and Removed

This section defines the way our model quantifies the first component of our model, which is the number of defects. Formal equations are proposed on how to calculate the number of estimated defects that would be found by a QA practice p in a single work product w or in a whole phase. The calculation of the estimated number of defects to be found is shown first followed by the calculation of the estimated number of defects to be removed.

In a defect detection activity, there are two main variables that are to be considered: 1- the Defect Removal Efficiency (DRE) value of the QA practice used and 2- the defect injection rate of the artifact exposed to the QA activity. The defect injection rate is an experience value which will be retrieved from our repository according to the data analysis and regression process discussed earlier. In our model, each work product of each phase has a specific defect injection rate value retrieved from past projects developed following the same approach. For more details see Section 4.9.3.

- Let I_w be the estimated injection rate of defects per KLOC in w of phase$_X$.
- Let N^{found} and $N^{escaped}$ equal the estimated number of defects found and escaped respectively by a QA practice p according to the QA practice's defect removal efficiency value DRE.

Assuming there are evenly distributed defects within a single work product[1], the estimated number of defects after applying a QA practice $p \in P$ is dependent on the estimated defect removal efficiency of that practice with respect to the category of the work product and the percentage (β_p) of the work product that is being inspected using this practice.

[1]If there is reason to believe this is not the case, the work product should be divided so that this assumption is reasonable for its parts.

5.3.1 Number of Defects Found

The first phase, the requirement *Phase*$_r$, will be taken as an example to show how our model works and how it can be generalised further to include the other phases later as was earlier depicted in the comprehensive model shown in Figure 5.3.

- **Number of Defects Found by a QA Practice** p

  The number of defects found or more precisely the number of estimated defects to be found in the work product w, relies entirely on the experience values given by the regression analysis process of previous projects (Section 4.9.2) and the estimated size of the work product currently under test (Section 4.4).

  Having determined these values, the estimated number of defects injected into a work product can be calculated by performing the multiplication of the defect injection rate value (I) of the work product by the estimated size of the work product $esize_w$ as follows:

$$eD_w = I_w * esize(w) \tag{5.1}$$

  The calculation of the mapping function $esize_w$ is previously explained in Section 4.4. Based on the eD value taken from Equation 5.1, the number of estimated defects that are going to be removed from the work product w by the QA practice p is as follows:

- **Number of Defects Found and Removed by the QA Practice** p

$$N^{Found} = \beta_p * eD_w * DRE_p \tag{5.2}$$

As can be seen from Equation 5.2, the estimated number of defects found is reliant on the DRE value of the chosen QA practice and the coverage ratio value (β) of the work product under testing assigned to that QA practice (Equation 5.2). Note that the choice of the QA practice is modelled using the variable β_p.

- **Number of Defects Found for Work Product w:**

  Referring back to our work product categorisation scenario discussed in Section 4.5, it was shown that in some cases an individual categorised work product may be assigned more than one single QA practice due to constraints like the short availability of time, the lack of testers who are familiar with such practices, etc. Therefore, in the case of the QA team applying more than one QA practice to a single work product, the estimated number of defects found for the whole work product equals the sum of estimated defects found by all QA practices applied to the work product based on their coverage ratio β. This is expressed in the following equation:

$$N_w^{Found} = \sum_{p \in P} \beta_p * eD_w * DRE_p \tag{5.3}$$

Note here that our model assumes that β is non-negative and the sum of the β should equal 1, $\sum\limits_{p \in P} \alpha_{p,w} = 1$. Therefore, to capture the case that parts of the work product are not verified, we assume $none \in P$ to be a special QA practice that has a defect removal efficiency of zero ($DRE_{none} = 0$) for all work products ($w \in W$). Likewise, the execution cost and execution time for applying the QA practice $none$ is also equal to zero as there is nothing to be verified.

◦ **Total Number of Defects Found in Phase$_x$**

For the whole development phase *Phase*$_(x)$, the number of defects found and removed can be calculated by extending the previous equations to include every work product as follows:

$$N_r^{Found} = \sum_{w \in W} \sum_{p \in P} \beta_p * eD_w * DRE_p \tag{5.4}$$

5.3.2 Number of Escaped Defects

From the previous equations for getting the number of found and removed defects by QA practices, it can be clearly noticed that what controls the number of found and escaped defects are the defect removal efficiency (*DRE*) values for the QA practices chosen. Accordingly, the same concept is followed as in Equations 5.2, 5.3 and 5.4 of estimated defects found and removed but with the substitution of the original variable of DRE with the the new variable (1-DRE). As a result, it will be possible to calculate the estimated number of unfound and removed defects using any QA practice applied as follows:

- **Number of Escaped Defects From a Single QA Practice:**

$$N^{Escaped} = \beta_p * eD_w * (1 - DRE_p) \tag{5.5}$$

- **Number of Escaped Defects From Work Product w:**

$$N_w^{Escaped} = \sum_{p \in P} \beta_p * eD_w * (1 - DRE)_p \tag{5.6}$$

- **Number of Escaped Defects From Phase$_x$ is:**

$$N_x^{Escaped} = \sum_{w \in W} \sum_{p \in P} \beta_p * eD_w * (1 - DRE)_p \tag{5.7}$$

5.4 Execution time and effort

5.4.1 Execution time

The execution time is defined in our model as the time required to run a QA practice on a software artifact in order to detect and remove its defects. As was mentioned in Section 5.2.5, the cost of running the QA practice needs to be quantified as a separate cost from the overall defect removal cost as in many cases software quality practitioners may find defects but may not remove them due to different constraints Section 3.4.

- Let t_p be the average execution time of applying a QA practice $p \in$ P to a specific work product $w \in$ W. The value of t_p is retrieved from the model's repository, as shown in Table 4.5 in Section 4.8 which will be normalised to a (Hour/FP) as time to size measure.
- Let size$_w$ be the size of a work product measured based on the artifact's unit of measurement.

The execution time for applying a QA practice is measured using the following equations:

- **Execution Time for a Single QA Practice** p

$$Ext_p = \beta_p * size_w * t_p \tag{5.8}$$

- **The Total Execution Time for Both** w **and the Overall Phase are Calculated Using The Following Equations Respectively:**

$$Ext_w = \sum_{p \in P} \beta_p * size_w * t_p \tag{5.9}$$

$$Ext_x = \sum_{w \in W} \sum_{p \in P} \beta_p * size_w * t_p \tag{5.10}$$

5.4.2 Execution Effort

Execution effort includes all types of cost needed to run a QA practice on a software artifact for the sake of finding defects and removing them. In our model, the execution effort value is expressed using a person/month scale. For the sake of conformance with the execution time aspect mentioned before and to unify the QA solutions resulting from the decision-making process, this value is converted to a person/hour scale.

As described earlier in Section 5.2.5, there are three interchangeable costs introduced by each QA practice p which in turn constitute the overall execution effort for the QA process.

The three types of cost are :

- **Cost of Execution** (Exc)
- **Cost of Defect Removal** (Rc)
- **Cost of Escaped Defects** (Esc)

The required equations for calculating the related effort for the three variables listed above are presented individually as follows:

1. **Cost of Execution (**Exc**):**

   During the application of a specific QA practice, there are two factors that contribute to the cost of executing a QA practice: the labour related cost and the practice related cost. With regard to the second factor, applying a QA practice usually entails costs related to setting up the tool, configuration etc., which is neglected in our model due to the difficulty in quantifying such costs and also due to the fact that such costs are considerably small.

In order to calculate the cost of the labour related factor required to execute a QA practice, a new variable needs to be introduced to quantify this cost; this variable is called the labour rate (Lr) which is a unit of money for a software QA practitioner who is assigned the mission of running and executing the QA practice. Using this variable, the execution cost can be calculated as a function of the execution time required to run the QA practice and the labour rate of the practitioner using it. It is assumed that the labour rate for all personnel involved in the QA execution process is the same. However, our model can also quantify different labour rates if this is not the case.

Exc is calculated by multiplying the labour rate (Lr) ,which is measured in any unit of money, for example (£/hour), by the execution duration retrieved from Equation 5.8 as follows:

$$Exc_p = Lr * Ext_p \tag{5.11}$$

2. **Cost of Defect Removal (**Rc**):**

   Unlike the cost of execution (Exc) which relies entirely on the time aspect, the cost of defect removal (Rc) depends on the number of defects the QA practice managed to uncover and their relative removal cost. Therefore, in order to calculate the value of Rc, a new variable, C_p, is introduced that refers to the cost of removing a defect through the QA practice p from the work product w. As was discussed in Section 4.8, our model will quantify the relative defect removal cost of each QA practice within the software development organisation with respect to the work product type and to the software development phase. This association is identified by the value of C_p which is going to be retrieved from our model repository as was earlier shown in Table 4.6.

- Let $C_p^{removal}$ refers to the cost of removing a defect originating from a $w \in$ W in phase $x \in$ X by $QA\ p \in$ P. The value of $C_p^{removal}$ is measured by a defect/hour scale.

The removal cost of applying the QA practice p is as follows:

$$Rc_p = \beta_p * eD_w * DRE_p * C_p^{removal} * Lr \tag{5.12}$$

3. **Cost of Escaped Defect Removal (Esc):**

The third cost aspect of the execution effort is the cost of escaped defects. In this case the number of defects that are estimated to escape from the development phase to the system testing phase is determined and then, the estimated cost required to remove them at that stage is calculated.

The future cost of any QA activity is important as it helps in the decision-making process of optimal QA solutions. In order to quantify this cost the variable C^{escaped} is devised which is equal to the average escalation factor of a defect that has escaped from a specific work product type and was discovered in the system testing phase. Here C^{escaped} is an experience value derived from past project data, capturing the impact of removing a defect in the system testing phases which belongs to and escaped from a specific work product which was much earlier in the development life cycle. An overview of the escaped cost calculation is depicted in Figure 5.7.

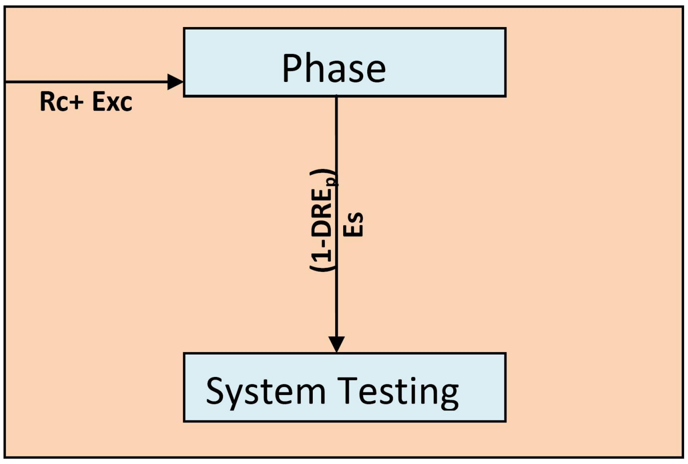


Figure 5.7: Aspects of Cost of the Model

Based on the previous description, the cost of fixing a defect escaped from a QA practice $p \in P$ and applied to a work product $w \in W$ which belongs to a phase $x \in X$ is :

$$Es_p = \beta_p * eD_w * (1 - DRE)_p * C_w^{escaped} * Lr \tag{5.13}$$

It should be noticed that the previous equation and the other equations of our model take into account the case that all or part of the work product is not assigned any QA practice and is not going to be verified. In this case the DRE_{none} expression, explained before in Section 5.3, is used to calculate the Esc value and will result in calculating the cost of all escaped defects estimated to have been originally injected into the work product under testing.

5.4.3 Combining QA Practices

In some cases and for software artifacts of a high significance, the project manager may like to apply more than one QA practice at once in order to reduce the defect injection rate in later phases. As was discussed earlier in Section 2.1.2, software QA practices differ in their efficiency and in their ability to find the specific scope of defects. In other words, defects found using a QA practice like formal inspection may differ from the defects that peer review can find, and vice versa.

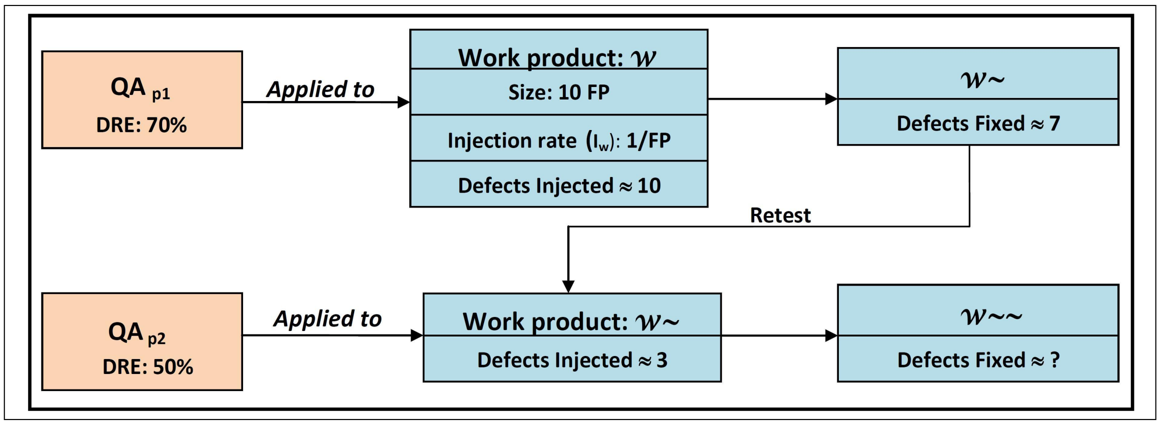


Figure 5.8: Combining QA Practices

In order to clarify the notion of applying more than one QA practice, an example is given as shown in Figure 5.8 that depicts a possible defect detection and removal scenario. As an example, the testing team chose to apply the QA practice p_1 to a specific work product w that has an estimated injection rate (I_w) of 1/FP, that is, 1 defect is injected into each functional point of the work product.

Assuming that the size of the work product is 10 FP, that means the work product is estimated to be injected with 10 software defects. The testing team applied a QA practice p_1, which was known to have a defect removal efficiency (*DRE*) = 70%, to the work product w. The estimated result of this activity was a new tested work product $\bar{w}$, which had approximately 7 defects that were fixed out of the original injected 10 defects.

70% * 10 = 7 defects found.

On the other hand, 3 defects are still injected into the work product $\bar{w}$ which will propagate to the later phases.

(1-70%) * 10 = 3 defects escaped.

In order to reduce the impact of these 3 escaped defects, the testing team decided to retest the same work product $\bar{w}$ using another QA practice p_2 which had a defect removal efficiency value $DRE = 50\%$ with respect to such a work product type. However, the defect injection rate of the work product $\bar{w}$ needs to be redefined according to the result of the first QA activity. The new injection rate of work product $\bar{w}$ is assumed to be 0.3 F/P as follows:

$$\boxed{I_{\bar{w}} = \frac{Defect\ injected \approx 3}{Size : 10\ FP} = 0.3}$$

Based on the values of $I_{\bar{w}}$ and the DRE of the QA practice $p_2 = 50\%$, the testing team would expect that the verification process would be estimated to reveal half of the injected defects. However, this optimisitic view of the second re-testing process may not always be correct; re-testing the work product with the QA practice p_2

may find no defects despite its defect removal efficiency value of DRE = 50%. The reason behind that is that in some cases the type of defects found by practice p_1 are the same defects found by practice p_2; therfore, applying QA practice p_2 on a work product which was already tested by practice p_1 may consume effort and time without tangible benefits.

This potential scenario is well-thought about in our thesis and we tried to tackle it by devising a new variable which was not included in our model which was described before in Section4.2.1. The devised variable will calculate the probability that a QA practice p_2 will find defects different to those defects found by the preceding QA practice p_1. This variable is called λ. An overview of this variable is depicted in Figure 5.9.

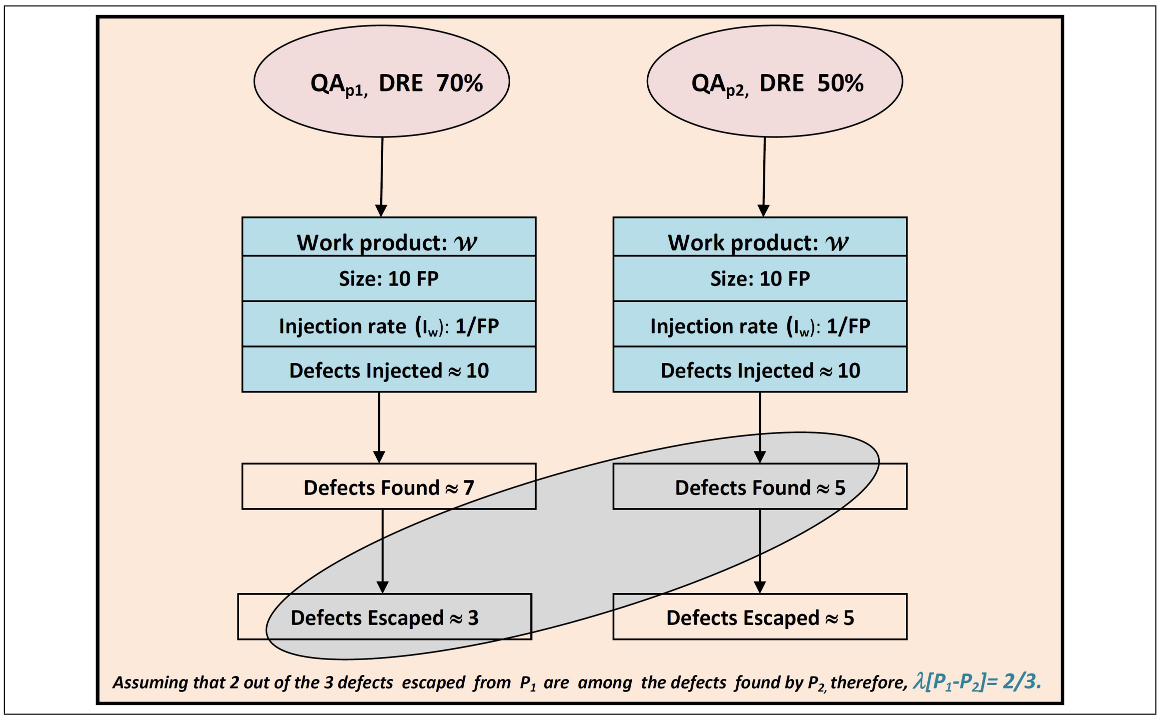


Figure 5.9: Lambda Variable

Based on Figure 5.9, the combination variable of applying the QA practice p_2 as a successor to the QA practice p_1 is $\lambda = 2/3$. That is, the QA practice p_2 is able to uncover 2/3 (two thirds) of the escaped defects from practice p_1.

By utilising the λ variable, the number of defects found after applying two QA practices sequentially is calculated by extending Equation 5.2 as follows:

$$N_w^{Found} = \beta_{p_1} * eD_w * DRE_{p1} + \beta_{p_2} * eD_w * (1 - DRE)_{p1} * \lambda_{p1-p2}. \quad (5.14)$$

$p_1, p_2 \in P$

5.4.4 Saved Cost (Sc) of QA Activity

Another cost variable introduced in our formal model and not considered a main part of our cost components mentioned before is the saved cost of a QA verification activity, referred to as Sc.

During the process of applying a QA practice, QA practitioners usually do not quantify a major important value that has a great influence in evaluating the current QA plan. This value is defined in our model as the saved cost which refers to the saving in cost of applying a QA practice which is expected to pay off later in the system testing phase. The main significance of this variable is that it is going to be utilised to evaluate the cost effectiveness of two potential QA plans in terms of their anticipated savings in the future compared with their current expected costs.

As mentioned before in the literature review presented in Chapters 2 and 3, removing software defects early in the SDLC yields to considerable savings in cost and time [78, 18, 65]. Therefore, these savings need to be quantified so as to utilise them in making the required trade-off process of current QA plans. In order to calculate the value of Sc, the impact escalation factor $C^{escaped}$ is used which was defined in Section 5.2.5.

Having the value of $C^{escaped}$ quantified, the saved cost for applying QA p with a weight β_p in work product w is :

$$Sc_p = \beta_p * eD_w * DRE_p * C^{escaped} \tag{5.15}$$

In fact Equation 5.15 is similar to Equation 5.2 regarding the number of defects for a specific QA practice. The only difference is that the estimated number of found defects is multiplied with the escalation cost factor C^{escaped} to get the cost of those defects if they manage to escape to the system testing phase.

5.5 Return on Investment and Total Development Cost

Defect detection and removal activities are considered to be an investment especially for profit-based software development organisations. This investment needs to be well-evaluated and studied to determine its positive and negative implications on the system development process.

Referring to the discussion in Section 3.5, the overall cost of any software development project is equal to the cost of the development activities and the cost of quality assurance practices implemented during the software life cycle.

Total development cost = Production development (COCOMO) + Cost of quality

As our focus in this thesis is mainly on the second aspect of cost, which is the cost of software quality ($CoSQ$), it will be shown how to evaluate the $CoSQ$ in a feasible way by using a well-known business measure, return on investment (*ROI*). In our model, the *ROI* measure is expressed according to a value to cost ratio as is shown in the following:

$$\boxed{ROI = \frac{Value}{Cost}}$$

- where :

Value: equals the savings cost of fixing defects found in the testing phase.

Cost: equals the effort of both executing the QA practice and fixing defects found.

First of all, the functions that are used in this research's model and constitute the model components are recalled, which are:

- **Execution Cost** (Exc)
- **Removal Cost** (Rc)
- **Escaped Cost** (Esc)
- **Saved Cost** (Sc)

In order to calculate the numerator for the *ROI* equation (Value), the following expression is used :

$$\boxed{\text{Value} = Sc - Exc + Esc + Rc}$$

For each QA p applied to a software artifact, the estimated value is identified by subtracting the estimated saved cost from the other three aspects of execution effort: execution cost, defect removal cost and escaped cost. On the other hand, the denominator in our ROI equation (Cost) can be defined as the sum of the three aspects of execution effort as follows:

$$\boxed{\text{Cost} = Exc + Esc + Rc.}$$

Accordingly, substituting the two values in our ROI equation yields:

$$\boxed{ROI = \frac{Sc - Exc + Esc + Rc}{Exc + Esc + Rc} \qquad (5.16)}$$

By generalising the previous equation to the whole work product w, the relative ROI of all QA plans applied to the work product w can be calculated by summing all variables as shown below:

$$\boxed{ROI_w = \frac{\sum_{p\in P} Sc_p - (\sum_{p\in P} Exc_p + Rc_p + Es_p)}{\sum_{p\in P} Exc_p + Rc_p + Es_p} \qquad (5.17)}$$

5.6 Summary

In any software QA plan, there are three aspects of cost that need to be well-quantified and measured. These aspects of cost consume a large share of the investment assigned to software quality activities. In this chapter, the formal components of our model were presented with respect to three cost aspects so as to enable a precise calculation of their values. This chapter defined the saved cost aspect resulting from applying QA practices and how this aspect could be utilised for trading off possible QA plan alternatives. A value-based component of our model was proposed that integrates Return On Investment (*ROI*) as a measure of the efficiency of a QA plans. The next chapter presents the implementation of our system by integrating the data collection and regression analysis with the mathematical components presented in this chapter.

Chapter 6

System Implementation

Objectives

- To implement our software quality system.
- To integrate our data collection and analysis components together.
- To show how to use the system as a quality management system.

6.1 Introduction

This chapter builds on Chapter 4, which explained the conceptual model and the different components of the system, and Chapter 5, which included the formal equations for all system components and how they connect and interact with each other.

6.2 System Implementation

The implementation of our system is achieved using ***C#*.Net** on a **Windows XP** operating system and **Microsoft Visual Studio** as a development environment. The main classes of our implemented system and their methods are illustrated in Figure 6.12. Given the complexity of applying the tool to all work products of a phase let alone all the SDLC phases, It was decided to limit the application of our tool to the work product of type *High* of the requirements phase so that the functionality of the tool is comprehended.

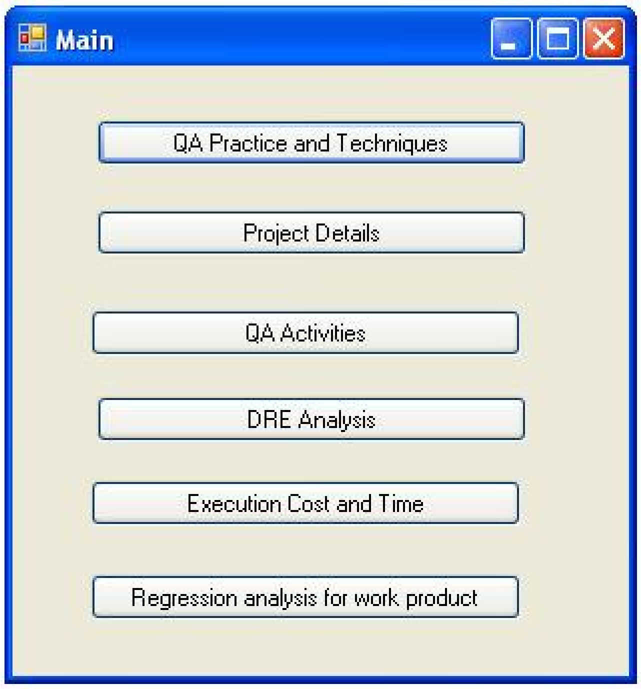
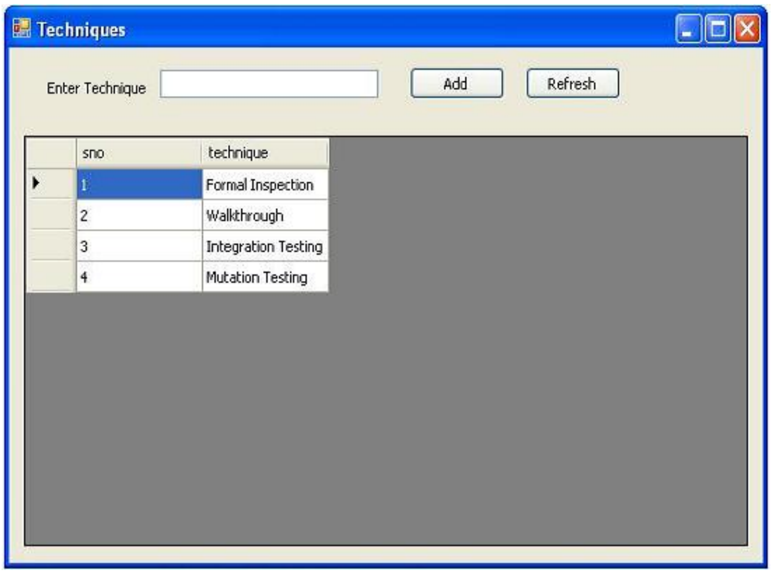


Figure 6.1: Tool User Interface

The user interface of our application is designed to include all the components of our system at once so that it is easily usable and gives the project manager and the QA practitioners the main points of control and interaction. An overview of the main application window is depicted in Figure 6.1.

6.2.1 The Entry of QA Practices

The first component of our application is the **QA Techniques** entry. In this stage, as is illustrated in Figure 6.2, software QA practitioners will enter the QA techniques and practices that are meant to be applicable to the requirements phase and are going to be utilised in the QA process for defect detection and removal activities. This list of practices will be recognised by our application once they are entered by the QA practitioner and are reserved a physical structure in the database of the system's repository. A description of the technique can also be added in a separate column next to its name to highlight its purpose.

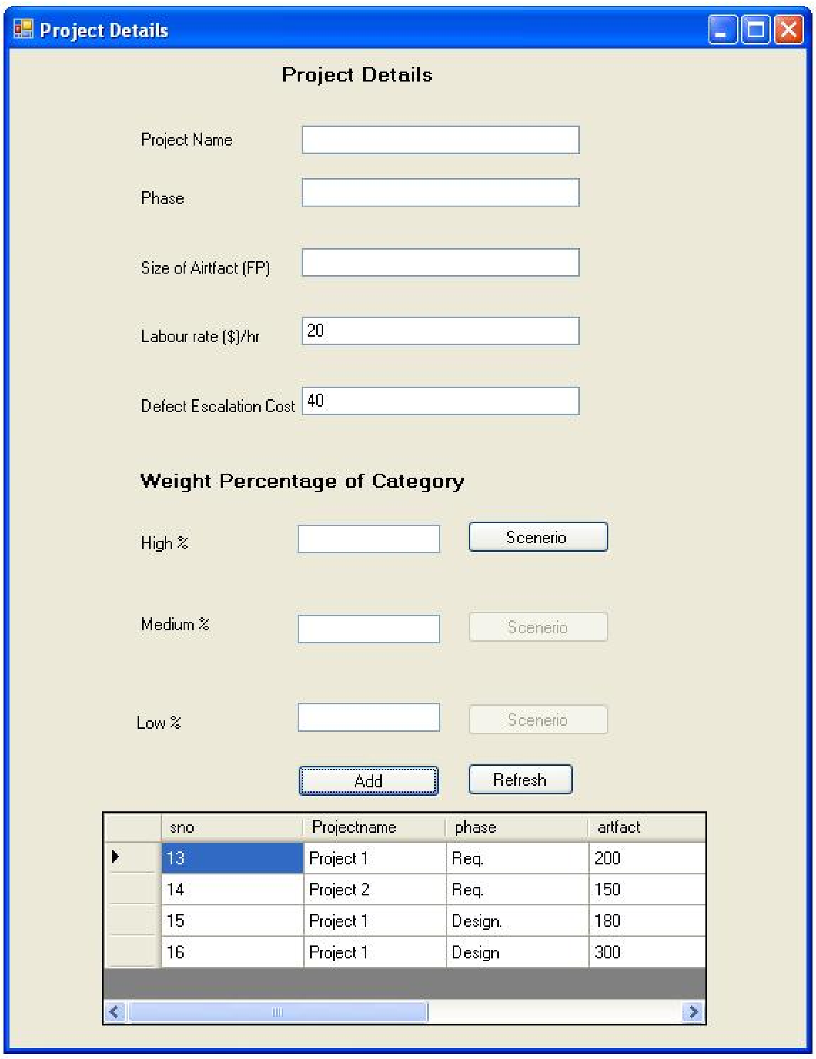


sno	technique
1	Formal Inspection
2	Walkthrough
3	Integration Testing
4	Mutation Testing

Figure 6.2: QA Techniques Entry Window

6.2.2 Project Detail Entry

The next component of our application is the **Project Details** entry process. At the beginning of any new software project development, QA practitioners will use the project details component of the application to store the details of the project under development (Figuer 6.3).


Project Details

Project Details

Project Name	
Phase	
Size of Airtfact (FP)	
Labour rate ($)/hr	20
Defect Escalation Cost	40

Weight Percentage of Category

High %	
Medium %	
Low %	

sno	Projectname	phase	artfact
13	Project 1	Req.	200
14	Project 2	Req.	150
15	Project 1	Design.	180
16	Project 1	Design	300

Figure 6.3: Project Details Input Window

Such project details are the phase name, total phase size, labour rate and defect escalation cost. Some of those entries like the estimated phase size and defect escalation cost are going to be verified later, after the completion of the software development process. In other words, once the software project is completed the size of the phase will be determined accurately and compared to the initial estimates of size given by cost estimation

models used like COCOMO II and then updated in the system repository.
This is also applicable to the defect escalation cost which will use an experimental value from past projects and will be updated later, on the completion of every new project.

6.2.3 Work Products Categorisation

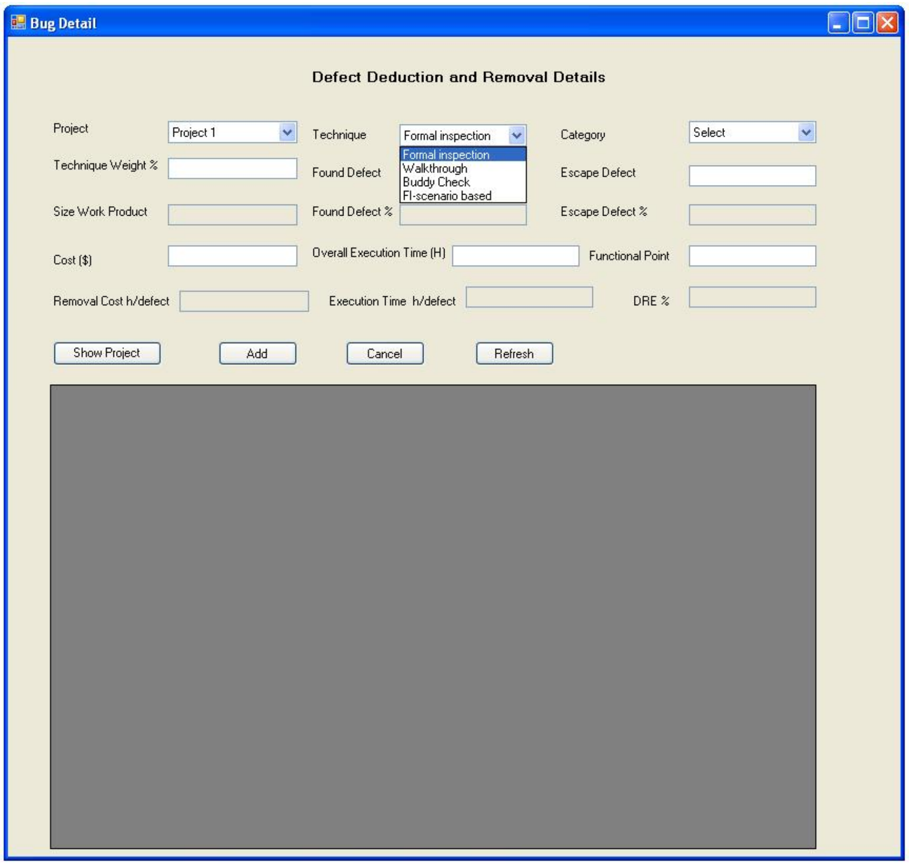
The second part that follows the project details entry process is the work products categorisation. Considering our application as a prototype for demonstrating the system functionality only, it is assumed that there are three levels of risk, *High, Medium and Low*, by which artifacts of software life cycle phases are categorised. Each category is assigned a weight expressed as a percentage value (%) constituting the total size of the phase deliverable. The mechanism of work products categorisation was previously detailed in Section 4.4.

The tabular form located below the main window of the project details (Figure 6.3) will include details of the previous projects that were developed following the same approach to show the depth of the repository with regard to the same phase. As was determined in Chapter 4, QA practitioners should apply the system initially to the first five projects so as to build enough of a repository of QA data to lend more accuracy and effectiveness to their decisions.

6.2.4 Details of QA Activity Entry

After defining the project name and risk weights for the phase's work product, the QA practitioner moves to the defect entry component of the tool. In this stage, relevant defect details resulting from current QA activities are entered and stored.

As shown in Figure 6.4, there are three pull-down menus for the project name, the number of QA practices defined before for that project and the work product categories. Note that the bug details window should include the phase name of the project, but as was clarified at the beginning of this chapter, our application is to demonstrate the functionality of our system in the requirements phase only.


Bug Detail

Defect Deduction and Removal Details

Project: Project 1
Technique: Formal inspection
Category: Select
Technique Weight %:
Found Defect:
Escape Defect:
Size Work Product:
Found Defect %:
Escape Defect %:
Cost ($):
Overall Execution Time (H):
Functional Point:
Removal Cost h/defect:
Execution Time h/defect:
DRE %:

Figure 6.4: QA Activity Details

The user assigns a coverage weight (%) value (β) for the QA practice chosen to perform the QA activity, and will then enter the number of defects found during the application of this QA practice.

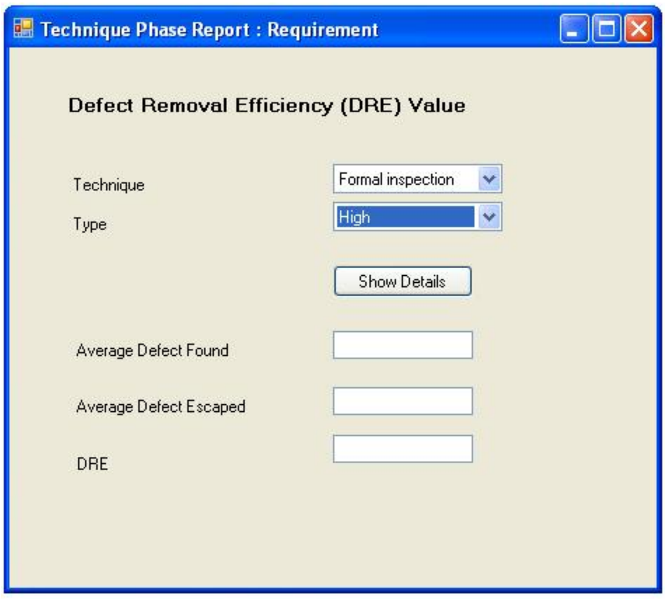
The number of escaped defects from the QA practices chosen will be updated later once the system testing phase completes. The entries of the shaded input boxes *(size of work* *product, % defects found, % defect escaped, etc.)*, are going to be calculated automatically based on the defects input details.

Moreover, the user should enter the execution duration required for applying the chosen QA practice; the input box ***Time(H)*** will include the overall duration of applying a QA practice with respect to the weight given. Then the ***Execution time*** box will calculate the estimated average duration time in an hour/functional point unit of measure. The cost of the defect removal process is entered in the input box *Cost(£)* and based on that, the tool will calculate the average cost per defect removal and show it to the user in the *Removal cost* box in a £/defect unit of measure. This measure is based on the labour rate value defined earlier in the project details component.

6.3 Data Analysis and Retrieval

6.3.1 Defect Removal Efficiency (DRE)

At the end of each project, or particularly at the system testing phase, data stored in the repository is retrieved for each work product, including all QA practices applied and resulting defects reports. This data is then analysed to get all of the needed values for the system variables. As is shown in Figure 6.5, the user can determine the DRE value for any QA practice used with respect to the work product it is applied to. The tool will compare the number of defects found to the number of escaped defects for all QA activities associated with a specific QA technique and hence calculate the average DRE value.


Technique Phase Report : Requirement

Defect Removal Efficiency (DRE) Value

Technique	Formal inspection
Type	High
Average Defect Found	
Average Defect Escaped	
DRE	

Figure 6.5: DRE of Work Products

6.3.2 Removal Cost and Time Determination

Following the same concept, the execution duration and defect removal cost associated with a QA practice with respect to a specific work product type is determined. The determination process is carried out by retrieving all data related to the cost of defect removal and average execution duration values from the repository (Figure 6.6). In our model, this step is done automatically and the project manager would have it ready as a separate component. An overview of this component is depicted in Figure 6.7.

6.3.3 Regression Analysis Component

The next component of the tool is the work product regression analysis (Figure 6.8). In this stage, regression analysis is carried out based on our conceptual model described in Section 4.9.2. This step is crucial in our model whereby the number of defects estimated to be injected into work products is calculated based on the defect injection rate

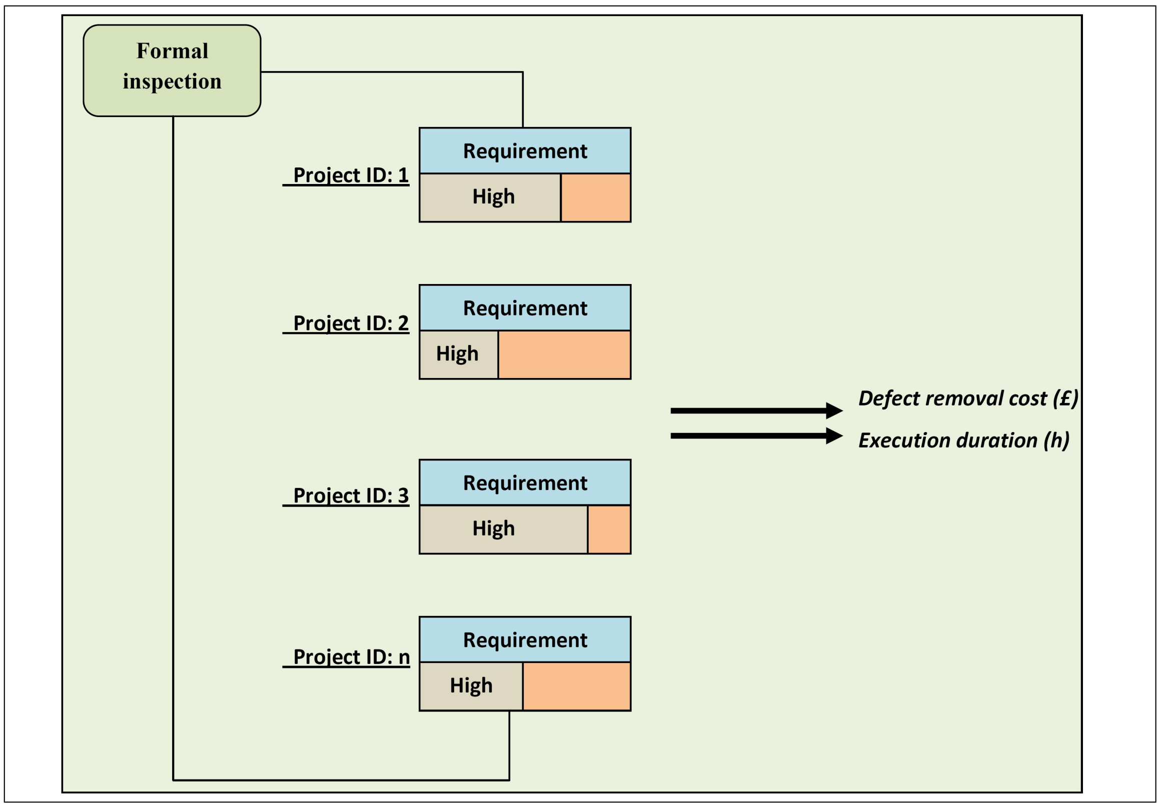
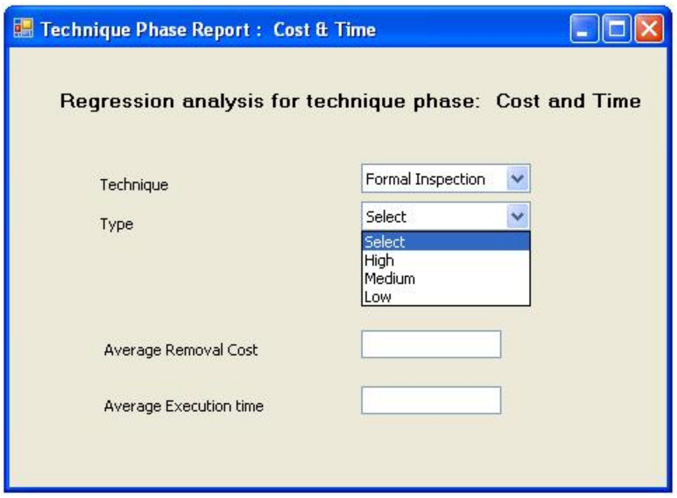


Figure 6.6: Average Values Determination



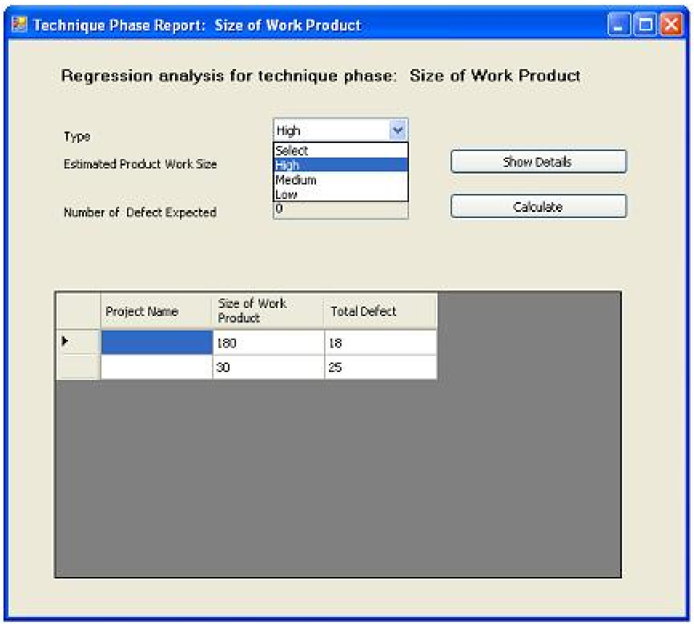
Regression analysis for technique phase: Cost and Time

Technique	Formal Inspection
Type	Select
Average Removal Cost	
Average Execution time	

]

Figure 6.7: Removal Cost and Execution Time Determination

per functional point scale.


Technique Phase Report: Size of Work Product

Regression analysis for technique phase: Size of Work Product

Type: High

Estimated Product Work Size

Number of Defect Expected: 0

	Project Name	Size of Work Product	Total Defect
▶		180	18
		30	25

Figure 6.8: Work Product Regression Analysis

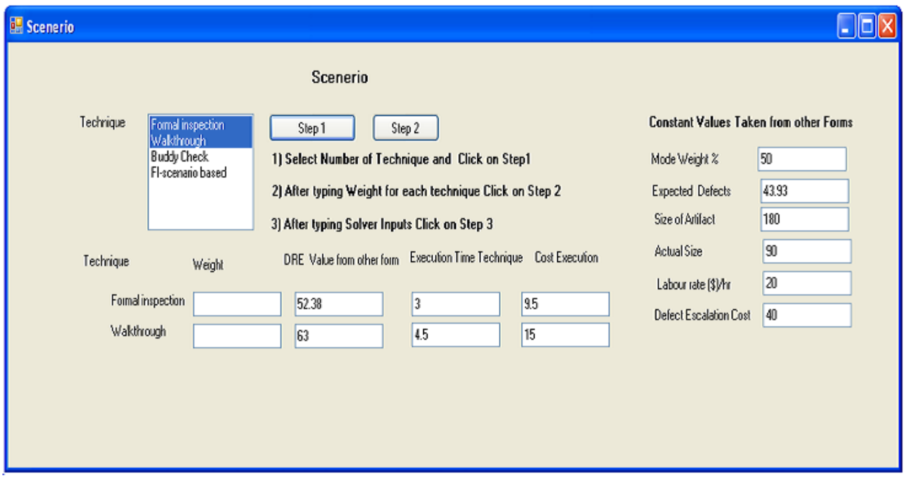
6.4 Building QA Decisions

After applying the model repeatedly to several software development projects following the same approach presented in this research, the data repository will increase and become more stable and reliable. Accordingly, the project manager can start using the model as an important support tool for building future decisions on QA plans. Back to our project details component shown in Figure 6.3, the project manager, after entering the new project details as usual, will click on the scenario button to get some informed estimates on the expected outcome of proposed QA activities. By clicking on the scenario button, a new window will pop up that shows the following details:

- QA techniques and practices available for the chosen work product, which in our example is *High*.

- The coverage weight value given to the work product (α) out of the total phase size.
- The size of the phase artifact and the actual size of the work product.
- The estimated number of defects injected into the work product according to the defect injection rate and the actual size.
- The defect escalation factor and the labour rate values.

An overview of scenario windows is depicted in Figure 6.9.



Scenerio

Technique: Formal inspection, Walkthrough, Buddy Check, FI-scenario based

1) Select Number of Technique and Click on Step1

2) After typing Weight for each technique Click on Step 2

3) After typing Solver Inputs Click on Step 3

Technique	Weight	DRE Value from other form	Execution Time Technique	Cost Execution
Formal inspection		52.38	3	9.5
Walkthrough		63	4.5	15

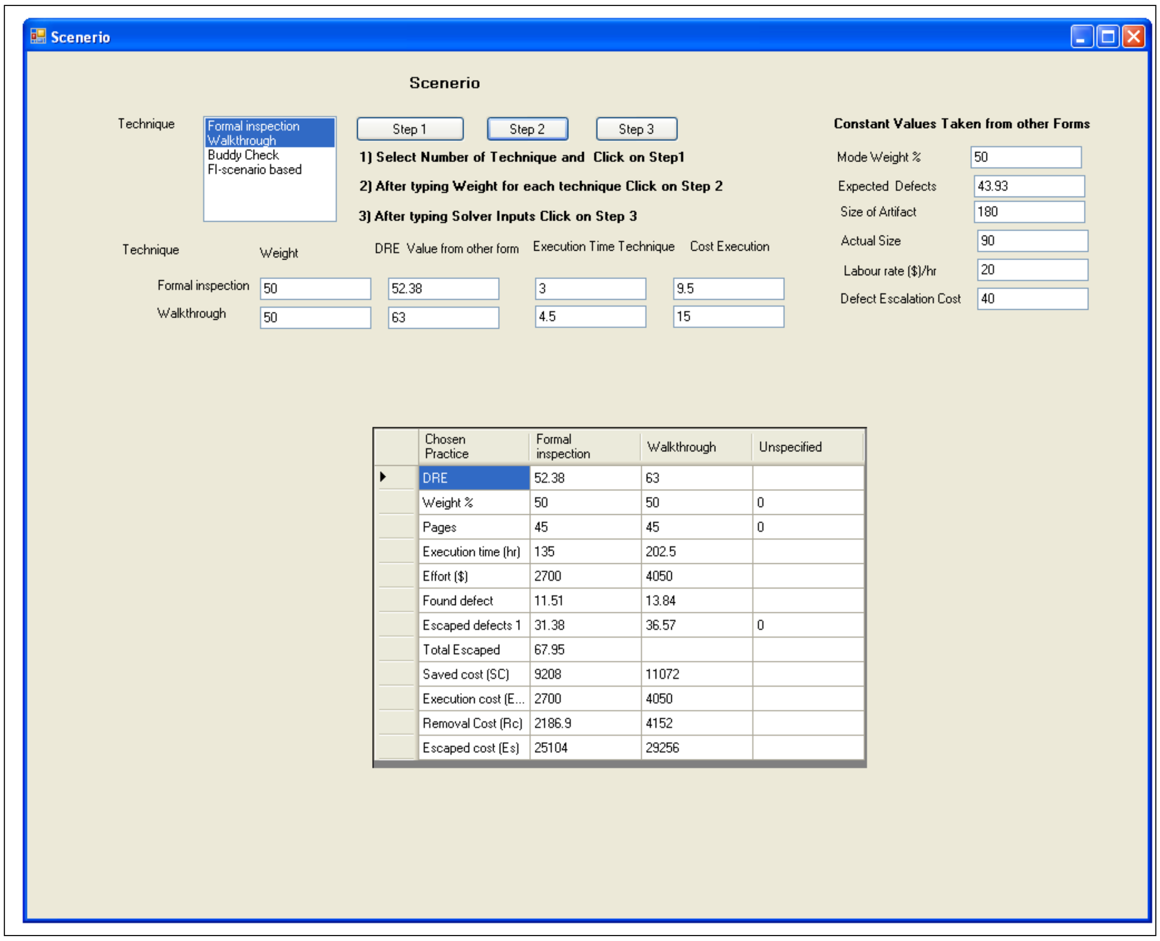
Constant Values Taken from other Forms

Mode Weight %	50
Expected Defects	43.93
Size of Artifact	180
Actual Size	90
Labour rate ($)/hr	20
Defect Escalation Cost	40

Figure 6.9: Scenario Making Component

Having all relevant details ready for the chosen work product, the project manager can choose a single practice or a group of QA practices from the listed available techniques to find out the possible outcome of the proposed defect detection and removal plan (Figure 6.10).

The choice of QA practices is subject to the project manager's insight and to the constraints associated with the project development process, the availability of testers familiar with an individual practice, the readiness of the practices, etc. Suppose the project manager chose to apply two QA practice to the work product *High*, a new form would appear with empty boxes which would require the weight value of each one of the two QA practices chosen. Along with weight input boxes, relevant experience details related to the two QA practices such as DRE, execution time and defect removal costs are automatically retrieved from the system repository. As discussed before, experience details are determined from previous software development projects.


Scenerio

Scenerio

Technique: Formal inspection, Walkthrough, Buddy Check, FI-scenario based

Step 1 | Step 2 | Step 3

1) Select Number of Technique and Click on Step1

2) After typing Weight for each technique Click on Step 2

3) After typing Solver Inputs Click on Step 3

Technique	Weight	DRE Value from other form	Execution Time Technique	Cost Execution
Formal inspection	50	52.38	3	9.5
Walkthrough	50	63	4.5	15

Constant Values Taken from other Forms

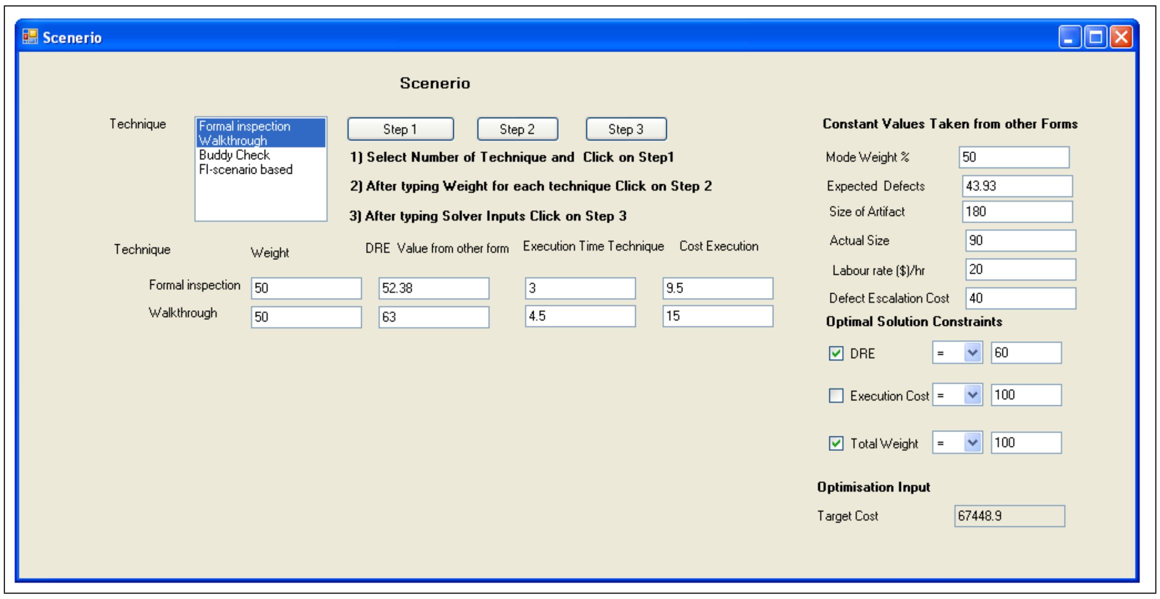
Mode Weight %	50
Expected Defects	43.93
Size of Artifact	180
Actual Size	90
Labour rate ($)/hr	20
Defect Escalation Cost	40

Chosen Practice	Formal inspection	Walkthrough	Unspecified
DRE	52.38	63	
Weight %	50	50	0
Pages	45	45	0
Execution time (hr)	135	202.5	
Effort ($)	2700	4050	
Found defect	11.51	13.84	
Escaped defects 1	31.38	36.57	0
Total Escaped	67.95		
Saved cost (SC)	9208	11072	
Execution cost (E...	2700	4050	
Removal Cost (Rc)	2186.9	4152	
Escaped cost (Es)	25104	29256	

Figure 6.10: Estimated Result of the QA activity

The next step after the weights determination is to get an overview of the expected outcome of the current QA practices with respect to the two practices chosen and the weights assigned to them. The system will perform the needed calculations following the equations discussed earlier in our formal model, and produce the result in tabular form. The results are detailed for every QA practice chosen such as the estimated number of removed and escaped defects, the execution cost and duration it requires, the overall defects removal cost and so on. The *Unspecified* column considers the situations in which part of the work product is left uninspected. Accordingly, the system will calculate all aspects of cost related to that part of the work product.

In Figure 6.11, the button labeled "**Step 3**" is the optimisation step of the system whereby an optimal solution is generated according to sets of conditions. Such conditions are the defect removal efficiency (DRE) value of the overall work product under inspection, the overall execution cost, removal cost, etc. The project manager can adjust the conditions (on the right of the figure) by entering their values and generating the solution. This solution will find the optimal distribution of weights that should be assigned to the two QA practices chosen at the beginning. The result of the optimal solution found will automatically change the values of weights given to the QA practices chosen.


Scenerio

Scenerio

Technique: Formal inspection, Walkthrough (selected); Buddy Check; FI-scenario based

Step 1 | Step 2 | Step 3

1) Select Number of Technique and Click on Step1

2) After typing Weight for each technique Click on Step 2

3) After typing Solver Inputs Click on Step 3

Technique	Weight	DRE Value from other form	Execution Time Technique	Cost Execution
Formal inspection	50	52.38	3	9.5
Walkthrough	50	63	4.5	15

Constant Values Taken from other Forms

Mode Weight %	50
Expected Defects	43.93
Size of Artifact	180
Actual Size	90
Labour rate ($)/hr	20
Defect Escalation Cost	40

Optimal Solution Constraints

✓ DRE	=	60
☐ Execution Cost	=	100
✓ Total Weight	=	100

Optimisation Input

Target Cost: 67448.9

Figure 6.11: Optimisation Step

6.5 Summary

The decision-making process regarding QA activities is subject to a lot of ongoing debate and different views due to the lack of a knowledge-based system to distinguish decision alternatives from each other. This chapter presented the implementation of our QA system tool and defined the main components of it. An outline was given to the formal steps of data entry and data analysis procedures. Also, this chapter showed how to utilise the optimisation function to find optimal solutions through the defect removal efficiency (DRE) required or on a cost of execution basis. The next chapter evaluates and assesses the functionality of our model and demonstrates its ability to make the trade-off process between QA plans on the basis of quality, cost and time.

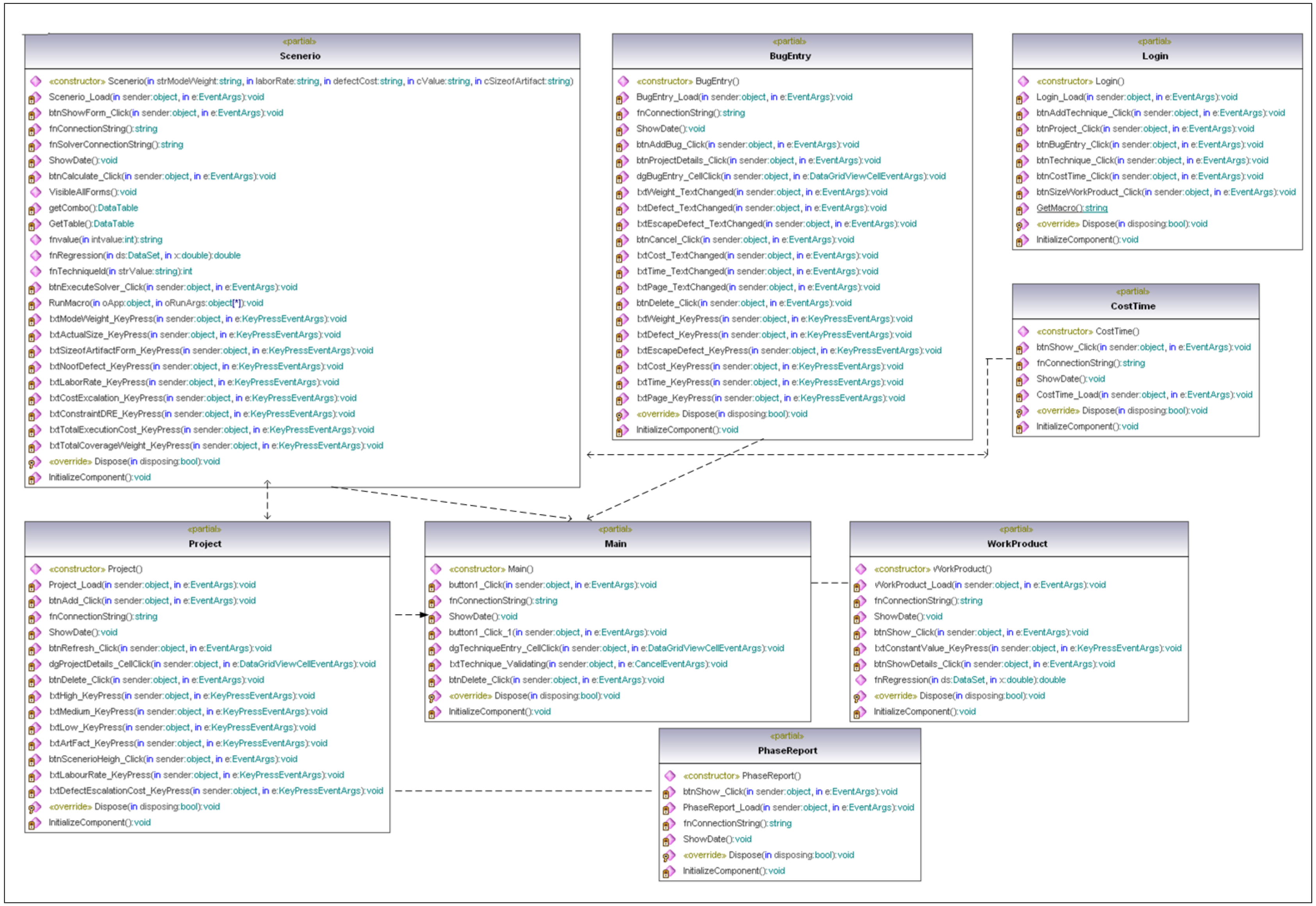

«partial»
Scenerio
«constructor» Scenerio(in strModeWeight:string, in laborRate:string, in defectCost:string, in cValue:string, in cSizeofArtifact:string)
Scenerio_Load(in sender:object, in e:EventArgs):void
btnShowForm_Click(in sender:object, in e:EventArgs):void
fnConnectionString():string
fnSolverConnectionString():string
ShowDate():void
btnCalculate_Click(in sender:object, in e:EventArgs):void
VisibleAllForms():void
getCombo():DataTable
GetTable():DataTable
fnvalue(in intvalue:int):string
fnRegression(in ds:DataSet, in x:double):double
fnTechniqueId(in strValue:string):int
btnExecuteSolver_Click(in sender:object, in e:EventArgs):void
RunMacro(in oApp:object, in oRunArgs:object[*]):void
txtModeWeight_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtActualSize_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtSizeofArtifactForm_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtNoofDefect_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtLaborRate_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtCostExcalation_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtConstraintDRE_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtTotalExecutionCost_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtTotalCoverageWeight_KeyPress(in sender:object, in e:KeyPressEventArgs):void
«override» Dispose(in disposing:bool):void
InitializeComponent():void
«partial»
BugEntry
«constructor» BugEntry()
BugEntry_Load(in sender:object, in e:EventArgs):void
fnConnectionString():string
ShowDate():void
btnAddBug_Click(in sender:object, in e:EventArgs):void
btnProjectDetails_Click(in sender:object, in e:EventArgs):void
dgBugEntry_CellClick(in sender:object, in e:DataGridViewCellEventArgs):void
txtWeight_TextChanged(in sender:object, in e:EventArgs):void
txtDefect_TextChanged(in sender:object, in e:EventArgs):void
txtEscapeDefect_TextChanged(in sender:object, in e:EventArgs):void
btnCancel_Click(in sender:object, in e:EventArgs):void
txtCost_TextChanged(in sender:object, in e:EventArgs):void
txtTime_TextChanged(in sender:object, in e:EventArgs):void
txtPage_TextChanged(in sender:object, in e:EventArgs):void
btnDelete_Click(in sender:object, in e:EventArgs):void
txtWeight_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtDefect_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtEscapeDefect_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtCost_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtTime_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtPage_KeyPress(in sender:object, in e:KeyPressEventArgs):void
«override» Dispose(in disposing:bool):void
InitializeComponent():void
«partial»
Login
«constructor» Login()
Login_Load(in sender:object, in e:EventArgs):void
btnAddTechnique_Click(in sender:object, in e:EventArgs):void
btnProject_Click(in sender:object, in e:EventArgs):void
btnBugEntry_Click(in sender:object, in e:EventArgs):void
btnTechnique_Click(in sender:object, in e:EventArgs):void
btnCostTime_Click(in sender:object, in e:EventArgs):void
btnSizeWorkProduct_Click(in sender:object, in e:EventArgs):void
GetMacro():string
«override» Dispose(in disposing:bool):void
InitializeComponent():void
«partial»
CostTime
«constructor» CostTime()
btnShow_Click(in sender:object, in e:EventArgs):void
fnConnectionString():string
ShowDate():void
CostTime_Load(in sender:object, in e:EventArgs):void
«override» Dispose(in disposing:bool):void
InitializeComponent():void
«partial»
Project
«constructor» Project()
Project_Load(in sender:object, in e:EventArgs):void
btnAdd_Click(in sender:object, in e:EventArgs):void
fnConnectionString():string
ShowDate():void
btnRefresh_Click(in sender:object, in e:EventArgs):void
dgProjectDetails_CellClick(in sender:object, in e:DataGridViewCellEventArgs):void
btnDelete_Click(in sender:object, in e:EventArgs):void
txtHigh_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtMedium_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtLow_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtArtFact_KeyPress(in sender:object, in e:KeyPressEventArgs):void
btnScenerioHeigh_Click(in sender:object, in e:EventArgs):void
txtLabourRate_KeyPress(in sender:object, in e:KeyPressEventArgs):void
txtDefectEscalationCost_KeyPress(in sender:object, in e:KeyPressEventArgs):void
«override» Dispose(in disposing:bool):void
InitializeComponent():void
«partial»
Main
«constructor» Main()
button1_Click(in sender:object, in e:EventArgs):void
fnConnectionString():string
ShowDate():void
button1_Click_1(in sender:object, in e:EventArgs):void
dgTechniqueEntry_CellClick(in sender:object, in e:DataGridViewCellEventArgs):void
txtTechnique_Validating(in sender:object, in e:CancelEventArgs):void
btnDelete_Click(in sender:object, in e:EventArgs):void
«override» Dispose(in disposing:bool):void
InitializeComponent():void
«partial»
WorkProduct
«constructor» WorkProduct()
WorkProduct_Load(in sender:object, in e:EventArgs):void
fnConnectionString():string
ShowDate():void
btnShow_Click(in sender:object, in e:EventArgs):void
txtConstantValue_KeyPress(in sender:object, in e:KeyPressEventArgs):void
btnShowDetails_Click(in sender:object, in e:EventArgs):void
fnRegression(in ds:DataSet, in x:double):double
«override» Dispose(in disposing:bool):void
InitializeComponent():void
«partial»
PhaseReport
«constructor» PhaseReport()
btnShow_Click(in sender:object, in e:EventArgs):void
PhaseReport_Load(in sender:object, in e:EventArgs):void
fnConnectionString():string
ShowDate():void
«override» Dispose(in disposing:bool):void
InitializeComponent():void

Figure 6.12: Main Classes of our System

Chapter 7

Evaluation of the System

Objectives

- To simulate the system functionality.
- To apply a hypothetical case study to our system to demonstrate its functionality.
- To signify the system's role as a decision support tool.
- To evaluate the efficiency of our model.

7.1 Introduction

The evolving nature of our system necessitates that the system should be piloted within a software development organisation to a few projects to build up its repository before it can be used as a decision support tool. However, taking into consideration the limited time allotted to do our research, it was difficult for us to implement our system for several projects developed in a single organisation in order to calibrate it within a specific domain of software projects. Moreover, because mid-sized and large software development projects, which our model mainly targets, last on average from 8 months to 1 year in terms of schedule, it was too difficult for us to conduct the required evaluation during our PhD research period.

In our endeavour to get some ready and real data to feed into our system, some connections were made with organisations that have software verification and validation activities like **NASA**'s[1] independent verification and validation department, and online software engineering warehouses like the **Promisdata**[2] repository. However, they responded that they had not got any defect data which conformed to our risk-based approach of work product categorisation.

Although this response was frustrating, it was at the same time positive because it showed the importance of our model and emphasized the originality of our work as none of those leading defects warehouses have such data available nor have they implemented a similar quality management system.

Another contact was made with Software Migrations Limited (SML), which is a UK-based software transformation company, to pilot our system in their software transformation projects. It is expected to start implementing our model with them after the completion of this research. The implementation of our system will be an additional service in a way that guarantees that our system will not affect their current QA activities.

In this chapter, fictitious data is used which depicts data of real QA activities of software development projects. This data will be used as an input to our system to simulate its functionality as an informative decision support tool for evaluating potential QA plans.

[1]NASA has an independent Verification & Validation program responsible for implementing QA best practices for complex software systems.

[2]A huge repository of empirical software testing experiments and QA activities.

7.2 Hypothetical Example

To show how our model functions, a scenario close to that of a software development project is given. Company x applied our model to manage and control their QA activities. They applied our model on a few software projects they used to develop (i.e of specific domain) and the application was in parallel with their QA activities they carried out with those projects. Having applied the model to several software development projects, they managed to build a considerable repository which contains QA data and rework reports of all QA activities of these past projects. The data is channelled and analysed in a way that conforms to our model, the thing that helped to determine necessary values for all of our model variables. Company x initiated a new software development project and the model would be used to help them get informed estimates on the expected outcome of their QA plans applied to this new project. This example will cover only the work product of type *High* in the software requirement phase for the sake of simplicity.

7.2.1 QA Process Details

The new software project that company x initiated is similar to all previous software projects and in line with the normal software domains which company x is well known for. The new project is estimated to be the size of 20000 KLOC based on readings of the effective cost estimation model the company implemented and is expected to run for over 2 months in schedule with an effort of 3000 person/month. As is the usual practice, the requirements phase of the SDLC goes through a categorisation process to determine the work product scheme that the phase should follow and to define the weights given to each work product according to the pre-defined categorisation levels.

The result of this categorisation process are 100 Functional Points (FP) constituting the work product of type *High*. The categorisation scheme of company x states that work products to be developed which are believed to hold high technical specifications of the software system need to be assigned a $High$ rating level. This rating level implies the high risk these work products represent to the software project.

Using our model, the software project manager of company x estimates that the work product $High$ is expected to be injected with 40 software defects given that the defect injection rate I = 0.4/FP. On the other hand, there are about nine QA practices that company x have previously applied to work products of the same category to perform the required defect detection and removal activities.

Data resulting from these activities was analysed and stored in the system repository and made ready for access and retrieval. However, in this project in particular, there are only three practices out of nine which are available to their team and proved their efficiency in inspecting similar work products. Those QA practices are:

- Formal inspection with scenario-based reading
- Formal inspection with ad-hoc reading
- Formal inspection with checklist-based reading

For more details about these practices see Section 2.5.2. The project manager retrieved data related to the three QA practices from the QA system and it showed the following details: for similar work products of type $High$, the scenario-based reading could reach a 75$\%$ defect removal efficiency (DRE = 75$\%$), the ad-hoc technique could reach about 69$\%$ (DRE = 69$\%$) and the checklist-based technique has a DRE = 50$\%$. In addition, other details related to the cost of defect removal and execution duration for each of the three practices with respect to the $High$ work product were retrieved from the QA system. A detailed overview of all details of the three practices is shown in Table 7.1.

Scenario-based reading technique		**Ad-hoc-based reading technique**		**Checklist-based reading technique**	
Variable	Value	Variable	Value	Variable	Value
DRE	75%	DRE	69%	DRE	50%
$C_p^{removal}$	(3 h/defect)	$C_p^{removal}$	(2.5 h/defect)	$C_p^{removal}$	(1 h/defect)
Ext	2 (h/FP)	Ext	0.5 (h/FP)	Ext	1 (h/FP)
$C_p^{escaped}$	40 (h/FP)	$C_p^{escaped}$	40 (h/FP)	$C_p^{escaped}$	40 (h/FP)

Table 7.1: QA Details For Three Techniques

7.2.2 QA Activity Scenario

Let us imagine the scenario that the project manager of the company x issued the QA team, before starting the defect detection and removal activities, with a task to achieve a DRE of 60% for the work product $High$. This task can be carried out using one or more of the available QA practices. Given the work product's criticality to the system, the project manager set another condition that this 60% is to be achieved as long as the whole work product is inspected, that is, each functional point needs to be tested by the QA team to reduce the injection rate of critical defects.

Based on that scenario and given the conditions set by the project manager, QA practitioners cannot fully assign the inspection process of the work product to one single technique from the three available QA techniques. The reason is that each of the three QA practices have a DRE value either more or less than the desired target of a DRE of 60%. Based on the foregoing, the QA team should make a wise decision so as to perform the QA process taking into accout the need to achieve the desired target of a 60% defect removal efficiency and full inspection coverage.

The rationale behind such constraints given by the project manger is that in some cases there is a difficulty and a potential risk to assign a single QA practice to perform the verification activities for the whole work product due to the fact that this QA practice may become unavailable at unknown time intervals during the QA process.

Another reason for the emergence of such constraints is related to the cost effectiveness aspect of the software development project and to how software development organisations realise this aspect differently. For many software development projects, especially those intended to generate profit, and with a high level of competition in the market, some software organisations have an alternative plan for their software development projects. The main step of this plan is to stop or reduce the time allotted for the software verification activities and release the software knowing that it has some defects. This decision results from the fact that releasing software earlier can allow the organisation to win a competitive advantage over their competitors, the thing that outweighs any benefit brought by requiring a delay to increase the software quality. Moreover, these defects can be resolved later in a new release of the software or exploited in one way or another to make more profit by selling maintenance contracts.

Considering the 60% DRE and with the help of our QA system, the project manager applied the necessary calculations relying on Equations 5.2, 5.4 and 5.6 and by trying to assign different weights of the three QA practices in a way to make the ultimate DRE value of the whole work product equal to 60%.

7.2.3 Estimated Result of the QA Process

As a result of these calculations, the project manager found many ways to reach a final defect removal efficiency (DRE) of 60% . These ways included either using a single practice out of the three available, two practices or all of them at once. Examples of these ways, or the QA plan alternatives found is shown in Table 7.2.

Option	Scenario-based reading	Ad-hoc-based reading	Checklist-based reading
1	30 %	18 %	52 %
2	10 %	40 %	50 %
3	0 %	53 %	47 %
4	20 %	25 %	55%

Table 7.2: Weight Distribution Alternatives

Looking at the four options of weight distribution of QA practices shown in Table 7.2, it can be clearly noticed that there are notable variations of weight values given to each one of the three QA practices. These variations are positive in the sense that they give the project manager and the QA team the flexibility to choose the option that best fits the current QA process and fulfills the QA goals.

However, it should be taken into consideration that each option from the above four introduces cost and time differently from the other options; therefore, the right choice of an option among the four alternatives should be made to verify not only the preferred targets but also to maintain the least amount of cost and time it is estimated to introduce. The QA team given this task should have a good estimate of the implication of each option on the overall cost and schedule of the project before committing to a particular option. The following subsection shows an example of the resulting calculations of cost and effort expected for the first option and summaries the results of the other options further.

7.2.4 1^{st} Option of Weight Distribution

The overall expected cost of applying the the QA process according to the 1^{st} option (30%, 18%, 52%) is as follows:

Technique	Weight	Defects found($\approx$)	Defects escaped($\approx$)
1	30%	9	3
2	18%	4	2
3	52 %	11	11
Total		24	16
DRE	60%		

Table 7.3: Estimated Number of Defects Found and Escaped ($\approx$) [3]

As can be seen from Table 7.3 the estimated total number of defects to be found are 24 defects. In this case and having considered that the estimated number of injected defects is equal to 40, the resulting DRE value for the work product is:

$$\boxed{\frac{9+4+11}{40} = 60\%}$$

In order to get the associated execution time and removal cost for applying the three practices to the chosen weights distribution, Equations 5.8, 5.9 and 5.11 are used to do the necessary calculations. Let us assume that the Labour rate (Lr) within company x equals £20 per working hour. Accordingly, the values of execution time (Ext) and execution cost (Exc) are as follows:

Option: 1		
Technique	Execution time (Ext)	Execution cost (Exc)
1	60 hours	£1200
2	9 hours	£180
3	52 hours	£1040

Table 7.4: Execution Time and Cost for Option 1

7.2.4.1 Removal & Escaped Cost

Depending on the cost aspects, discussed in Section 5.4.2, which defined three categories of cost, the values of the other two costs need to be calculated along with the execution cost calculated before. These two categories of cost are the defect removal cost (Rc) and the cost of escaped defects or the escaped cost (Esc). The calculation of such costs are based on Equations 5.12 and 5.13 as depicted in the following table.

Option: 1		
Technique	Cost of defect removal Rc * Lr	Cost of escaped defect Esc * Lr
1	£540	£2400
2	£248	£1785.60
3	£208	£8320

Table 7.5: Rc & Esc for Both Techniques

Accordingly and based on Tables 7.4 and 7.5, the total cost which would be introduced

[3] The number of defects are represented as integers, and rounding floating numbers to integers may slightly change the results of the calculations shown in this section, since the numbers of defects in our example are considerably small.

as a result of applying the three practices would be the sum of their relative cost variables: Exc, Rc and Esc. The results of the calculations are depicted in Table 7.6.

Option: 1			
Technique	Execution cost (Exc) + Removal cost (Rc)	Escaped cost (Esc)	Total
1	£1740	£2400	£4140
2	£428	£1786	£2214
3	£1248	£8320	£9568
Total	£3416	£12506	£15922

Table 7.6: Total Cost

For the other three options of weight distribution shown in Table 7.2, the same calculations are applied to calculate the overall cost introduced by each option and summarise the final values. An overview of the final result is shown in Table 7.7.

All options			
Option	Execution cost (Exc) + Removal cost (Rc)	Escaped cost (Esc)	Total
1	£3416	£12506	£15922
2	£2703	£12768	£15471
3	£2389	£12778	£15167
4	£3075	£12880	£15955

Table 7.7: Total Cost for all Options

7.2.5 Comparing QA Options

Having completed the calculations for all of the four options, it would seem that options vary in their costs of execution, removal and escaped cost. The reason that these variations are to some extent slight is due to the fact that variables of each QA practice such as the

DRE, $C^{escaped}$ and $C^{removal}$ are close to each other and also due to the fact that the size of the work product is not large enough to make the differences in these values notable and influential. It can be clearly noticed from **Table 7.7** that the third effort distribution of 0%, 53% and 47% yeilds the best overall result of *£15167* compared with *£15922*, *£15471* and *£15955* for options 1, 2 and 4 respectively. However, comparing alternatives of QA plans manually or on an individual basis is cumbersome and ineffective in a rapid software development environment where time is crucial. Another driver for the ineffectiveness of such manual comparison is due to the fact that there might be other better solutions which were not thought about. The list of QA weight options shown in Table 7.2 is just an example of possible weights and does not cover all the available options.

Another scenario which needs to be taken into account while comparing QA plan alternatives is the stability of the development process and its internal and external environments. In some cases it does not always mean that the best QA plan is the plan that yields the least cost among other available plans; there are other issues which need to be taken into consideration to favour one option over another.

For example, if we compared the 1st and the 3rd options, we will notice that overall the 3rd option is more cost-effective than the other options with the saving of a cost difference of £755. However, usually when developing a software project that may take 2 to 3 years of the life cycle time, variations on the available budget are a common problem during any stage of the project development process, and the project manager cannot accurately determine or anticipate them.

For instance, in our example of inspecting the work product $High$, let us suppose that the current budget available for performing the QA process within the phase is £3500. This will cover the cost resulting from executing the QA practice Exc and the defect removal cost Rc.

In addition, the budget that would be available at the system testing phase to cover any escaped defects from this work product would be less than £12,550. Depending on these given resources, the project manager may in that case go for the 1st option instead of the more cost-effective 3rd option as the budget that would be available for the escaped cost would be less than the necessary £12,778 in the 3rd option while there would be enough of a budget to cover the costs of Exc and Rc in the 1st option.

7.3 Dealing With Constraints

Software development is a strenuous and risky process as there are many constraints that may be faced by the project manager at periodic intervals during the software life cycle phases [93, 2]. Such constraints hinder project managers from planning the development activities properly. These constraints may be previously determined at the beginning of the SDLC or they may emerge accidentally at any development phase without prior notice. Accordingly, the project manager should be able to handle these constraints and be aware of them during the software development process. Examples of these constraints include schedule reduction, budget shortage, lack of testers, the unavailability of a preferred QA technique etc.

In the following example it will be shown how effective our model is when dealing with such constraints and how it can give the project manager the flexibility and the proactivity to control them. For the sake of simplicity we chose one single constraint as an example which is budget shortage.

7.3.1 Budget Shortage Constraint

Referring to our example of company x as is shown in Table 7.6, let us assume that the available budget during the development process for inspecting the $High$ risk work product is £3000, which includes running the QA process and the defect removal activities. This amount should be invested in a way to maintain high DRE and minimise both execution and escaped costs.

A few scenarios will be worked out which will cover two points:

1. ***Investing the whole amount in a scenario-based reading technique only.***
2. ***Investing part of the amount in ad-hoc reading and saving the difference.***
3. ***Using scenario-based and ad-hoc reading techniques jointly.***
4. ***Investing the whole budget in an ad-hoc reading technique.***

In all scenarios, the necessary calculations will be made to compare the results. An overview of the needed values for the scenario-based and ad-hoc-based reading techniques is shown in Table 7.8.

Scenario-based reading technique		Ad-hoc reading technique	
Variable	Value	Variable	Value
DRE	75%	DRE	50%
$C_p^{removal}$	(3 h/defect)	$C_p^{removal}$	(1 h/defect)
Ext	2 (p/h)	Ext	1 (p/h)
$C_p^{escaped}$	40 (p/h)	$C_p^{escaped}$	40 (p/h)

Table 7.8: Scenario-based and Ad-hoc based Reading Techniques

- **1st Scenario:** *Investing the whole amount in a scenario-based reading technique* *only.*

Technique	Weight	Execution time (Ext)	Execution cost (Exc) & Removal cost (Rc)	(Esc)
Scenario-based	100%	200 hours	£5800	£8000

Table 7.9: Scenario-based Outcome

As shown in Table 7.9, applying the scenario-based reading technique with a full weight coverage of 100% would generate £5800 of Exc and Rc. This value exceeds the available budget of £3000 assigned to the QA plans of the work product. In order to have the QA plan comply with the available budget, the project manager should try to decrease the overall cost of the practice used by reducing the weight value assigned to it.

Based on this decision and using Equation 5.11 and 5.12 which showed that with £3000 of Exc and Rc, the weight value given should be roughly only 52% out of the whole weight. The relative Exc and Rc for the estimated weight is shown in Table 7.10.

- **2nd Scenario:** *Investing part of the amount in ad-hoc reading and saving the difference.*

Technique	Weight	Execution time (Ext)	Execution cost (Exc) & Removal cost (Rc)	(Esc)
Ad-hoc based	52%	104 hours	£3016	£4160

Table 7.10: Execution & Removal Cost of 52% of Work Product $High$

As shown in Table 7.10, only 52% of coverage ratio is given to the QA scenario-based reading technique; that is, 48% of the work product will be left uninspected which is, in turn, injected with defects that may propagate to the system testing phase. The estimated escaped cost of the remaining 48% of the work product is calculated in order to quantify the consequences of leaving the rest of the work product uninspected. An overview is shown in Table 7.11.

Technique	Weight	Escaped Defect (Esc)	Escaped cost (Esc)
None	48%	19	£15200

Table 7.11: Escaped Cost of 48% of Work Product

As shown in Table 7.11, the Esc soared to £15,200 which would have a negative impact on the budget of the later phases and the project manager may not favour such a scenario. To reduce the escaped cost, the project manager may decide to apply the ad-hoc based reading technique jointly with the scenario-based reading technique to both stay within the budget allowance and to mitigate the effect of the Esc value. This is clarified in the third scenario.

- **3rd Scenario:** *Using scenario-based and ad-hoc reading techniques jointly.* Using our QA system and based on Equations 5.13, 5.12, 5.11 and **5.6**, the project manager worked out many weight distributions of applying the scenario-based reading technique and the ad-hoc reading technique. However, as stated before, the total Exc and Rc for both techniques should adhere to the constraint that it does not exceed the £3000 of the QA budget. The project manager found another option to handle this constraint in a way that the scenario-based reading would be reduced to 20% and that the remaining 80% would then be assigned to a checklist-based reading. An overview of the result is shown in Table 7.12.

Technique	Weight	Found defects	Escaped defects	(Exc)&(Rc)	(Esc)
Scenario-based reading	20%	6	2	£1160	£1600
Checklist-based reading	80 %	16	16	£1920	£12800
Total	100 %	22	18	£3080	£14400

Table 7.12: Weights Distribution of Scenario-based and Ad-hoc Reading Techniques

- **4th Scenario**: *Investing the whole budget in the ad-hoc reading technique.*

  This is in case the project manager decides to exclude the scenario-based reading technique from the QA plan due to its high Exc value and assign the whole QA process to the ad-hoc reading technique with a weight of 100%. The expected result of this weight distribution is illustrated in **Table 7.13**.

Technique	Weight	Found defects	Escaped defects (Esc)	(Exc) & (Rc)	(Esc)
Ad-hoc reading	100%	20	20	£2400	£16000

Table 7.13: Ad-hoc Reading Technique QA Process

It would seem that from the last scenario, £2400 only of the available budget was invested in the Exc and Rc which yielded an overall saving of £600. This saving can be used later to reduce the total value of Esc as follows:

$$\boxed{\text{£16000 - £600 = £15400 } Esc}$$

However, by comparing the previous four scenarios, it can be clearly noticed that the 3rd scenario (scenario-based: 20%, ad-hoc based: 80%) outweighs the others in terms of its overall low cost and in terms of the resulting DRE value. For example, the net *Esc* value of the 3rd scenario is £14,400 compared with £15,400 for the 4th scenario. Accordingly, the resulting savings for the benefit of the 4th scenrio calcualted by subtracting one value from the other is:

£15400 - £14400 = £1000

7.4 Joining QA Practices

As was clarified in Section 5.4.3 of our mathematical model, the project manager may prefer to expose some software artifacts to an extensive QA process including more than one QA practice instead of a single QA process. The incentive behind such a decision is the known criticality of the inspected artifact to the software development phase it belongs to or to the whole software life cycle. A new variable, λ, was introduced that refers to the probability of the QA practice x to uncover escaped defects through the application of another QA practice y.

In the following example, it will be shown how to exploit such a variable in evaluating re-testing decisions of a work product using another QA practice. This example will use the scenario-based and ad-hoc based reading techniques whose details were shown in Table 7.8.

Let us assume that the scenario-based reading technique was used for the QA activity of the work product $High$ with a full weight coverage (ie. $\beta = 100\%$). As shown in Table 7.14, the work product $High$ is estimated to be injected with 40 defects and the scenario-based reading technique is expected to find 30 defects out of the original 40 defects, therefore, 10 defects are likely to escape to the system testing phase.

Based on this scenario, the relative cost associated with using this QA practice includ-

Technique	Weight	Defects found	Defects escaped
Scenario-based	100%	30	10

Table 7.14: Defects Found & Escaped Using Scenario-based Technique

ing the Ext , Exc, Rc and Esc is calculated. An overview of these costs are summarised in Table 7.15.

Technique	Execution time (Ext)	Execution cost (Exc)	Rc	Esc
Scenario-based	200 hour	£4000	£1800	£8000

Table 7.15: Relative Cost of Scenario-based Technique

By considering the escaped cost (Esc) only, as shown in Table 7.15, assigning the scenario-based reading technique to the QA process generated £8000 of an Esc value as a result of the 10 escaped defects. As a result of that, the project manager has to make a decision, either to re-inspect the work product again with another QA practice in order to reduce the leakage of estimated escaped defects (Esc), or to invest the remaining budget assigned into inspecting other work products. The significance of the λ variable will be shown in one situation: that is re-inspecting the whole work product with another technique, assumed here to be ad-hoc reading, and comparing the benefits gained from the re-inspection process with the cost invested.

Assuming that the value of λ between an ad-hoc reading technique and a scenario-based reading technique is equal to 0.6, this means the ad-hoc based technique can uncover 60% of the escaped defects through applying a scenario-based reading. Using our QA system and by applying Equations 5.9 and 5.11, the implication of applying an ad-hoc reading technique in terms of its execution time (Ext) and execution cost (Exc) values needs to be calculated. An overview of the calculation results is shown in Table 7.16.

Technique	Weight	Execution time (Ext)	Execution cost (Exc)
Ad-hoc reading	100%	100 hours	£2000

Table 7.16: Ad-hoc Reading Technique Ext & Exc Values

As can be noticed from Table 7.16, the value of Rc, which points to the removal cost, is excluded due to the fact that the ad-hoc reading technique is dealing with a partially corrected version of the work product. Accordingly, to get the new value of Rc, defects which were successfully detected and removed by the previous QA practice (scenario-based) are to be eliminated. The original defect removal efficiency value DRE of the ad-hoc reading technique, which in our example is 50%, needs to be replaced with the value of λ as the new defect removal efficiency with respect to the previously tested work product. Therefore:

Technique	Old DRE	New DRE
Ad-hoc reading technique	50%	60%

Now, using the new DRE value (60%) for the ad-hoc reading technique and applying it to the same work product according to **Equation 5.14**, the number of estimated defects to be found and its relative Rc value can be calculated. An overview of the results is shown in Table 7.17.

Technique	β	Found defects	Rc
Ad-hoc reading technique	0.6	6	£120

Table 7.17: Ad-hoc Reading Technique Rc Value

Updating the value of Rc in Table 7.16 results in the following :

Technique	Ext	Exc	Rc
Ad-hoc reading technique	100 hours	£2000	£120

Table 7.18: Ad-hoc Reading Technique Cost Values

Having completed the calculations, the project manager is now able to compare the findings of both decisions by grouping the total cost and the escaped cost for the senario-based reading technique with the additional cost of the ad-hoc reading technique. An overview of the comparison is shown in Table 7.19.

Decision	QA practices	Execution & Removal cost	Escaped cost
1	Scenario-based only	£5800	£8000
2	Scenario & Ad-hoc-based techniques	£7920	£3200

Table 7.19: Total Cost and Escaped Cost of Both Decisions

The value of Esc of the 2nd decision, which is running the ad-hoc reading technique after the scenario-based technique, could be reduced to £3200 giving a net benefit over the 1st decision of almost £5000. On the other hand, the total cost of the 2nd decision, which represents the execution and removal cost, increased to £7920 with a £2000 increase over the scenario-based technique's total cost.

At first glance, it would seem that it is a wise decision to run the ad-hoc reading technique sequentially after the scenario-based reading technique as there are considerable savings in the total cost of Rc, Exc, and Esc of more than £2680.

$$\boxed{(£5800 + £8000) - (£7920 + £3200) = £2680}$$

However, such a comparison needs to be computed automatically without the need to compare the investment with the revenue for each cost aspect in every decision. Moreover, the result of the comparison should be based on a scientific standard so as to make it persuasive to both the project manager and the software stockholders. To compare the results of Table 7.19 from an economic perspective, the project manager should apply the return on investment (ROI) principle to measure the efficiency of the investment in QA plan alternatives. The mathematical representation of the ROI that shows the relationship between our model's cost aspects is described in Equation 5.17 in Section 5.5.

7.4.1 Return On Investment (ROI) of QA plans

In order to calculate the *ROI* value of any QA activity, the project manager needs to know the saved cost (Sc) for both decisions along with the other cost aspects. Back to our discussion in Section 5.4.4, the saved cost (Sc) value is an important factor in comparing decision alternatives and is going to be utilised along with the *ROI* principle. Therefore, the saved cost needs to be quantified for every QA decision to make the necessary trade-offs between QA plan alternatives.

Based on Table 7.19 and relying on Equation 5.15, the Sc value of implementing a scenario-based reading technique for the 1st decision only and scenario-based & ad-hoc-based techniques for the 2nd decision is calculated as follows:

$$\boxed{Sc_1 = 100\ \% * 75\ \% * 40 * 20 = £24000}$$

$$\boxed{Sc_2 = Sc_1 + 100\ \% * 60\ \% * 40 * (1\text{-}75\ \%) * 20 = £28800}$$

The values of Sc_1 and Sc_2 are substituted in the return on investment equation which is :

$$ROI = \frac{Sc - Exc + Esc + Rc}{Exc + Esc + Rc}$$

Based on the substitution process, the return on investment for both QA plans will be :

- 1^{st} Decision

$$\boxed{ROI_1 = \frac{24000 - 4000 + 1800 + 8000}{4000 + 1800 + 8000} * 100\% = 74\%}$$

- 2^{st} Decision

$$\boxed{ROI_2 = \frac{28800 - (4000 + 2000) + (1800 + 120) + 3200}{(4000 + 2000) + (1800 + 120) + 3200} * 100\% = 160\%}$$

It would seem that the investment in the second decision of re-inspecting the high work product with the ad-hoc reading technique after the scenario-based reading technique is estimated to yield 86% more of a saving than the saving that would result from using the scenario-based reading technique alone.

$$\boxed{160\% - 74\% = 86\ \%}$$

Therefore, the project manager may go for the second decision as it gives better value than the first decision.

This scenario shows the importance of our devised metric λ as an essential variable for defining the cost effectiveness of a QA decision to re-test an artifact using another QA practice. It also signifies the novelty of our QA system in terms of its dynamic decision-making process that helps software managers get informed estimates on their QA activities on a value-based basis.

7.4.2 Optimisation Method

It was discussed in Section 7.2.5 that there are many options of weight distributions that can be assigned to the three QA practices: *scenario-based, ad-hoc-based and checklist-based* reading techniques so that the final defect removal efficiency equals 60%. Therefore, trading off these options in terms of their overall cost and their ability to fuilfil given constraints should be carried out accurately and automatically.

In any software development project, the project manager always seeks the optimal solution which maintains the least amount of cost and effort. There are many optimisation techniques that can be applied in this case to get the optimal solution required. In this research, the Linear Programming - Simplex Optimisation Method is used to solve our example in Section 7.2.3. It will be assumed that the calculations were preformed before do not exist and hence the project manager does not know what the options that are available for the current QA plan are and what cost each option would entail.
The available QA techniques: scenario-based, ad-hoc based and checklist-based will be referred to with x, y and z respectively in our optimisation model.

First of all, for any optimisation problem, we need to define the three parts of the optimisation model which are type, target variable(s) and constraint(s) as follows:

- **Optimisation type**

  It is clearly seen from the scenario given above that we want to achieve an estimated DRE target of 60% with minimal cost, that is, it is an optimisation problem of minimising cost. However, QA cost as defined in Section 5.4.2, consists of three types: the cost of execution (Exc), the cost of removal (Rc) and the cost of removing escaped defects (Esc). Accordingly, the cost type to be minimised out of the three types needs to be determined. In this optimisation example, we chose to include all the aspects of cost, and hence the optimisation type is to minimise the overall cost resulting from the QA process carried out by any distribution of the three QA practices.

- **Target function(s)**

  In this part we need to define the target functions that our model will be applied to based on the optimisation type described above. Having already mentioned that our model type is to minimise the overall cost of the proposed QA plan, functions that contribute to the overall cost need to be defined for each one of the three QA practices.

  - **Escaped cost (Esc)**

    By applying Equation 5.13 of Section 5.4.2 to the three practices we have:

    $Esc_x = \beta_n * 40 * (1 - 75\%)_x * 40 * 20$

    $Esc_y = \beta_n * 40 * (1 - 69\%)_x * 40 * 20$

    $Esc_z = \beta_n * 40 * (1 - 50\%)_x * 40 * 20$

    Total.Esc $_{High}$ = Es_x + Es_y + Es_z

  - **Execution effort.**

    Execution effort includes the execution time and the defects removal cost.

$$Ext_{total} = \sum_{p \in x,y,z} = \beta_p * size_w * t_p$$

$$Exc_{total} = \sum_{p \in x,y,z} = Lr * Ext_p$$

Total-eff. = $Ext_{total} + Exc_{total}$

- **Changing Variables**

  Changing variables are those which will be changed by the target functions of the model to find the optimal solutions. In our example, where the overall cost needs to be minimised, it can be clearly noticed that the main cost driver that affects this cost is the number of defects found and escaped which in turn depends on the weight assigned for each QA practice.

- **Constraints**

  The constraints that the project manager sets are the target defect removal efficiency ($DRE = 60\%$) and the full coverage ratio. In other words, all artifacts of the work product should be inspected maintaining a DRE value of 60% at the least cost. This can be intrepreted formally as follows:

  1. Minimise Total.Esc + Total.eff.

     Subject to:

  2. $\beta_x + \beta_y + \beta_z = 1$
  3. $DRE = 60\%$

In order to resolve this optimisation problem, the **LINDO system version 11.1** (LINDO Systems, Inc) [4] was used. An overview of the source code of our optimisation model mod-

[4]The LINDO system is a powerful tool to solve optimisation models; linear, non-linear, etc. It has a built-

elled on the LINDO system is shown in Listing 7.1.

in language to express the optimisation problems. Lindo can be downloaded from http://www.lindo.com.

```
  MIN TotaleffX + TotaleffY + TotaleffZ + EscX + EscY + EscZ
  // Objective function minimise overall cost.

    SUBJECT TO

        ExtX - 200 WGHX = 0
        ExtY - 50 WGHY = 0
        ExtZ - 100 WGHZ = 0
NdefectsX - 30 WGHX = 0  // Number of defects Eq. for practice (x).
NdefectsY - 27.6 WGHY = 0
NdefectsZ - 20 WGHZ= 0
         NdefectsX + NdefectsY + NdefectsZ <= 24
         // Constraint given to fulfil the DRE target.

RcX - 1800 WGHX = 0 // Removal Cost Eq. for practice (x).
RcY - 1380 WGHY = 0
RcZ - 400 WGHZ= 0

ExcX - 20 ExtX = 0
ExcY - 20 ExtY = 0
ExcZ - 20 ExtZ = 0
       TotaleffX - ExcX - RcX = 0
       TotaleffY - ExcY - RcY = 0
       TotaleffZ - ExcZ - RcZ = 0



EscX - 8000 WGHX = 0 // Escaped cost Eq. for practice (x).
EscY - 9920 WGHY = 0
EscZ - 16000 WGHZ = 0
```

```
WGHX + WGHY + WGHZ = 1
// Coverage weights of the three practices equls 1.


END
```

Listing 7.1: Source Code of The Optimisation Model

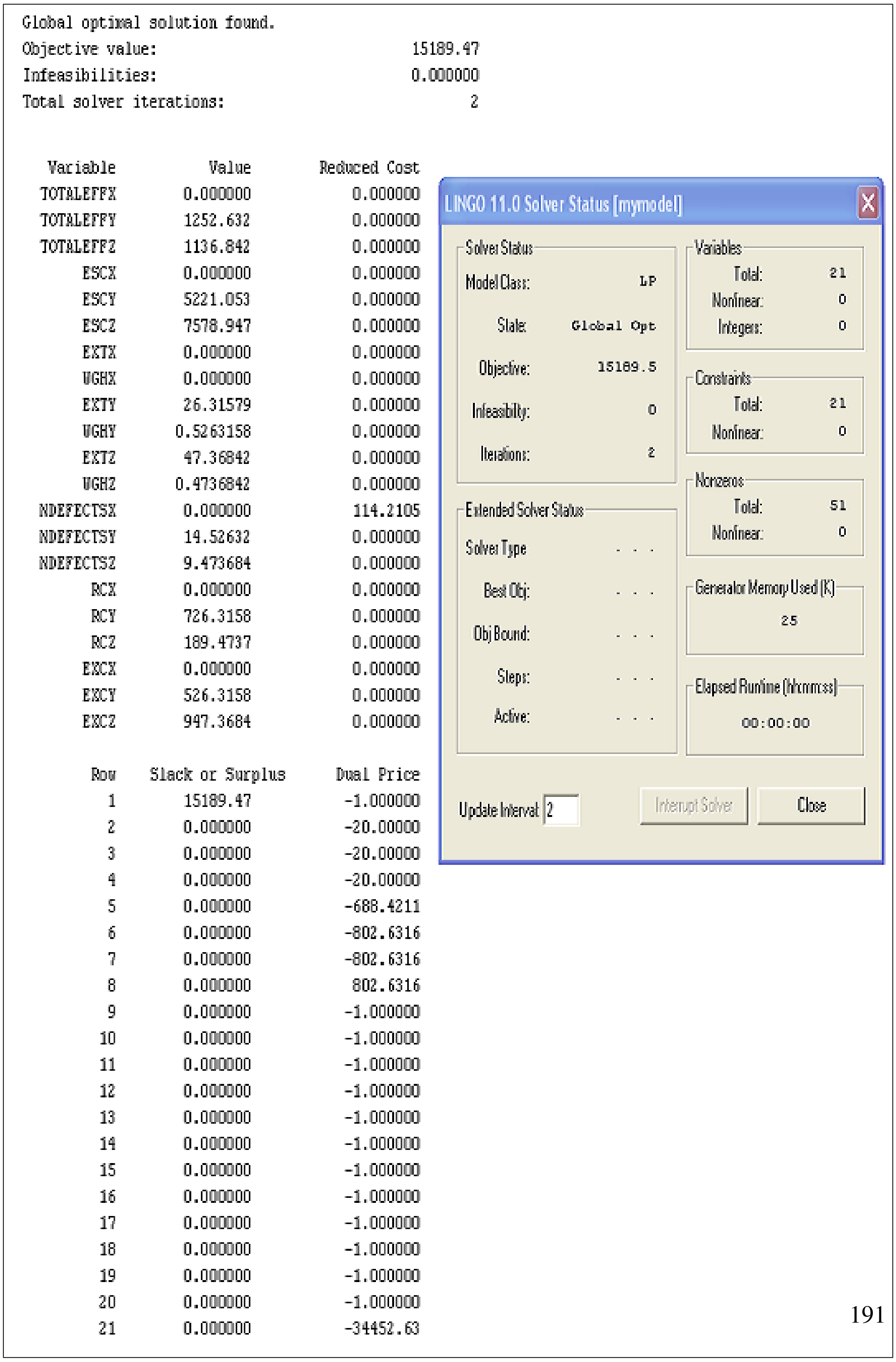
After running the linear programming model, the LINDO system found an optimal solution to the objective function of our model which equals $\approx$ 15189.47. This value can be achieved by having the following values of weight coverage :

WGHX = 0 WGHY $\approx$ 53%, WGHZ $\approx$ 47%

Matching those values to our example means that the scenario-based reading technique should be excluded from the inspection process and that the checklist-based reading and ad-hoc based reading techniques would be assigned the weight values of 53% and 47% respectively so that the overall cost of the QA process is minimal and the final DRE $\approx 60\%$. An overview of the optimisation model result is depicted in **Figure 7.1**.

7.5 Possible Issues with Our Models

The application of our model will follow the structure proposed in this thesis and is expected to provide encouraging and helpful QA estimates. However, some issues may arise upon the application of our system which may have an effect on its final results. In this section, examples of such problems will be shown and what the corrective actions for them are.



```
Global optimal solution found.
Objective value:                              15189.47
Infeasibilities:                              0.000000
Total solver iterations:                             2


      Variable           Value        Reduced Cost
     TOTALEFFX        0.000000            0.000000
     TOTALEFFY        1252.632            0.000000
     TOTALEFFZ        1136.842            0.000000
          ESCX        0.000000            0.000000
          ESCY        5221.053            0.000000
          ESCZ        7578.947            0.000000
          EXTX        0.000000            0.000000
          UGHX        0.000000            0.000000
          EXTY        26.31579            0.000000
          UGHY       0.5263158            0.000000
          EXTZ        47.36842            0.000000
          UGHZ       0.4736842            0.000000
     NDEFECTSX        0.000000            114.2105
     NDEFECTSY        14.52632            0.000000
     NDEFECTSZ        9.473684            0.000000
           RCX        0.000000            0.000000
           RCY        726.3158            0.000000
           RCZ        189.4737            0.000000
          EXCX        0.000000            0.000000
          EXCY        526.3158            0.000000
          EXCZ        947.3684            0.000000

           Row    Slack or Surplus      Dual Price
             1        15189.47           -1.000000
             2        0.000000           -20.00000
             3        0.000000           -20.00000
             4        0.000000           -20.00000
             5        0.000000           -688.4211
             6        0.000000           -802.6316
             7        0.000000           -802.6316
             8        0.000000            802.6316
             9        0.000000           -1.000000
            10        0.000000           -1.000000
            11        0.000000           -1.000000
            12        0.000000           -1.000000
            13        0.000000           -1.000000
            14        0.000000           -1.000000
            15        0.000000           -1.000000
            16        0.000000           -1.000000
            17        0.000000           -1.000000
            18        0.000000           -1.000000
            19        0.000000           -1.000000
            20        0.000000           -1.000000
            21        0.000000           -34452.63
```

```
LINGO 11.0 Solver Status [mymodel]

Solver Status                      Variables
  Model Class:          LP           Total:       21
  State:        Global Opt           Nonlinear:    0
  Objective:       15189.5           Integers:     0
  Infeasibility:         0         Constraints
  Iterations:            2           Total:       21
                                     Nonlinear:    0
Extended Solver Status             Nonzeros
  Solver Type:       . . .           Total:       51
  Best Obj:          . . .           Nonlinear:    0
  Obj Bound:         . . .         Generator Memory Used (K)
  Steps:             . . .               25
  Active:            . . .         Elapsed Runtime (hh:mm:ss)
                                         00:00:00

Update Interval: 2        Interrupt Solver    Close
```

Figure 7.1: Optimisation Model Solution

7.5.1 Deviation Control of QA Process

In any software development organisation, factors like a stable level of process improvement, team cohesion, development flexibility etc., contribute to maintaining an efficient software life cycle which conforms to the organisation's pre-defined plans and baselines. However, there are a few cases of software development projects in which deviation from the normal baselines may occur due to unexpected development problems.

One of the issues, which is of interest to us, is the defect detection and removal efficiency of their QA activities. Accordingly, the project manager should have the ability to notice any deviations arising within the software under development so as to be able to make an early decision in order to bring the deviated software back to its normal state. As our QA system relies entirely on the manipulation of previous QA data stored in the repository, any non-conformity between the data of the QA activity of the current software and the average values stored in the repository is raised and shown to the project manager to make informed decisions related to that QA activity.
In order to clarify this point, an example of the deviation control process, implemented in our model, will be shown and how valuable it is to both the project manager and the overall software cost, time and quality.

Let us assume that the project manger of company x discussed in Section 7.2 chose to apply a scenario-based reading practice to inspect the work product of type $High$ and with a full coverage value of $\beta = 100\%$. Using our model and performing the required calculations to estimate the number of defects found and escaped using the chosen QA practice, the estimated result is shown in Table 7.20.

As can be seen from the table below, there are 40 defects expected to be injected in 100 functional points of the software requirements specification (SRS) document; that is,

Size of SRS(FP)	Type of work product	Size(FP)	Estimated Defects	QA Technique	Weight	Estimated Found Defects
150	High	100	40	Scenario-based reading	100 %	30

Table 7.20: Initial Overview of the QA Activity

the defect density is approximately 0.4/functional point.
The QA assurance team initiated the inspection process using the scenario-based reading and relying on the estimated results given by our system; however, after inspecting 15% of the work product the QA activity generated the following QA data result:

Type of work product	Inspected so far	Size (FP)	Found Defects
High	15 %	22.5	10

Table 7.21: Result of 15% of The QA Activity

Based on the results shown in Table 7.21, inspecting 15% of the work product using the scenario-based reading technique uncovered 10 software defects out of the 40 defects that were estimated to be injected with in the work product.
Therefore and by a simple calculation, the expected number of defects injected into the whole work product can be obtained as being equal to:

$$\boxed{\text{Total defects} = \frac{10 * 85\%}{15\%} + 10 \approx 67 \Rightarrow}$$

Based on the estimated total defects found, the expected defect density for the work product $High$ is 0.67/FP. This estimated defect density considerably exceeds the original estimated defect density given by our system which was 0.4/FP.

According to the scenario mentioned above, the QA team can realise that the current work product does not conform to the average values derived from previous projects and there must be some issues which have caused it to deviate from the normal baselines. There are three options that would be proposed to handle this situation. The first option is to carry on inspecting the work product using the same QA practice. The second option is to replace the QA practice with other low-cost QA practices such as the checklist-based technique. A last resort would be to rewrite the whole work product. Using our system, the estimated outcome of the 1stoption and 2ndoption is summarised as follows:

1. **Proceeding With the Existing QA Process**

   First, the total cost is calculated if the project manager decided to carry on with the current QA activity using the same QA practice, and the consequences of this decision would be as follows :

Found defects	Escaped defects	Rc & Exc	Esc
50	17	£7000	13600 £

Table 7.22: Overall Cost of The 1stOption

2. **Rewriting the Whole Work Product**

   On the other hand, that project manager could go for the decision to stop inspecting and rewrite the whole SRS document. Let us assume that the developer labour rate (Lr) is £30/hour. Also, assuming that each FP costs about 1.5 h/FP, the result will be as follows:

Cost of Work Product Development (hour/FP)	Expected size (FP)	Developer labour rate (£/hour)
1.5	100	£30

Table 7.23: Cost of Rewriting The Work Product $High$

Based on **Table 7.23**, the expected cost of rewriting the whole work product is :

$$\boxed{1.5 * 100 * 30 \approx £4500}$$

Now, this cost with the cost of execution and defect removal need to be sum up as a result of applying the QA process again using Equation 5.11, 5.12 and 5.13 as follows:

Estimated found defects	Exc & Rc	Esc
30	£5800	£8000

Table 7.24: Overall Cost of the 2ndOption

According to Table 7.24, the total cost associated with the work product $high$ to re-write the whole SRS and re-inspect the work product again is :

$$\boxed{5800 + 4500 = £10300}$$

Based on the estimated results of the two options shown in Tables 7.22 and 7.24, it can be clearly noticed that the second decision of terminating the existing QA process would bring a higher value to the project than the value introduced by the first decision. Regardless of the cost effectiveness of the two options, the scenario shows how important our model is in maintaining a detailed and accurate baseline of any QA process carried out within the software development organisation and how these baselines can be utilised to monitor any non-conformance during the software life cycle.

7.5.2 Using Statistical Process Control

The last section showed the ability of our model to make corrective actions to any deviation or non-conformity of the current QA process to the normal baselines stored in the system's repository. In order to avoid the recurrence of the previous situation what is called " a preventive action " needs to be implemented which is an approach by which an alert is given to the QA team for any none-conformance to the organisation's baselines. The Statistical Process Controls (SPC) was chosen to be applied for that purpose.

The benefits of SPC in our model are twofold:

- It uncovers extreme changes in average values.
- It discovers weaknesses of the quality inspection process.

Table 7.25 depicts details of QA activities that were applied to the work product $High$ in the requirements phase, these details were captured from many past software projects.

Project-ID	Phase	Work product type	Size (FP)	F. defects	Es. defects	T.defects	DRE
1	Req.	High	120	35	8	43	81.4 %
2	Req.	High	90	35	14	49	71.4 %
3	Req.	High	45	9	6	15	60 %
4	Req.	High	168	38	9	47	80.9 %
5	Req.	High	180	40	9	49	81.7 %
6	Req.	High	18	5	5	10	50 %
7	Req.	High	29.7	15	1	16	93.57 %

Table 7.25: QA Details of Past Software Projects

Based on the data shown in Table 7.25, informed baselines can be obtained of defect injection rates for the work product for every project as follows:

Project-ID	Work product size	Number of defects	Defect Density (I)
1	120	43	0.4
2	90	49	0.5
3	45	15	0.3
4	168	47	0.28
5	180	49	0.27
6	18	10	0.55
7	29.7	16	0.53

Table 7.26: Defect Injection Rates of Past Projects

From the above two tables, the software organisation can establish the statistical process control and define the control limits for the DRE, total defects etc.

to calculate the control limits we need to get the following variables: mRi, Clx, UCLx and LCLx.

- mRi refers to moving average and can be calculated by getting the absolute value of subtraction of adjacent values. mRi = |Xi - Xi-1| i = 2 . . . n
- Clx is the average value of a set of values.
- UCLx is the upper limit of a set of values UCLx= X + 2.66 * mR
- LCLx is the lower limit a set of values LCLx = X - 2.66 * mR

We will apply this approach to Table 7.26 of injection rate values.

Based on Table 7.27, the : CLx, UCLx, and LCLx values can be calculated as follows:

UCL_x	CL_x	LCL_x
0.7	0.4	0

Project-id	Defect Density (I)	mRi
1	0.4	0
2	0.5	0.19
3	0.3	0.21
4	0.28	0.05
5	0.27	0.01
6	0.55	0.28
7	0.53	0.02
$m\bar{R}i$		0.13

Table 7.27: mRi Values of Defect Injection Rates

By utilising these values, a statistical process control chart can be established (Figure 7.2) where the average defect injection rate is the centerline surrounded by the upper limit and lower limit (dotted lines).

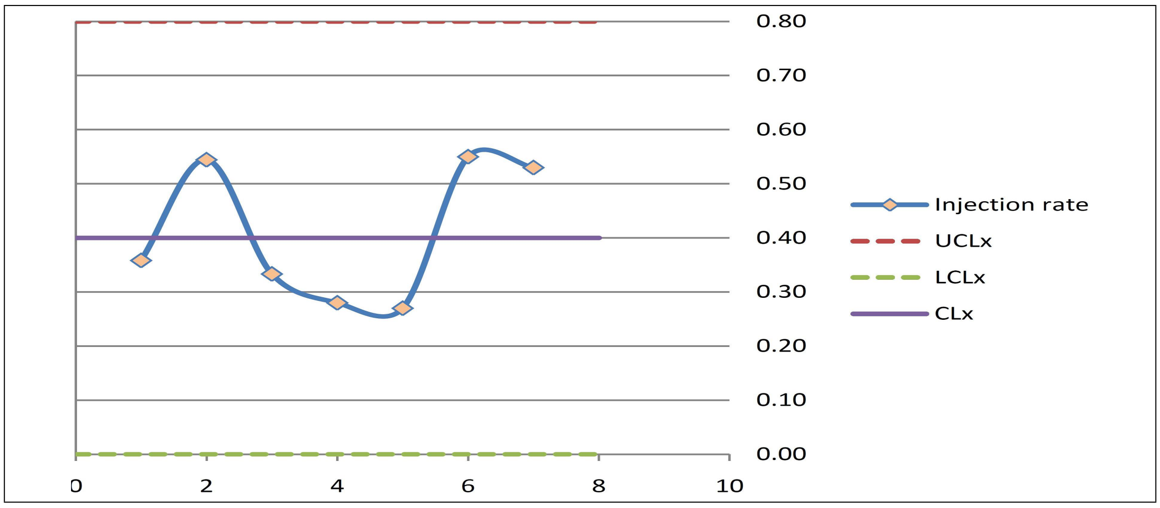


Figure 7.2: Control Limits Chart

According to the figure above, the QA team will be able to define any non-conformance to the defect density rate of any new work product. The new defect density of new work products should adhere to the normal baseline CLx with acceptable deviations that do not exceed the upper and lower control limits of the charts. This approach can be also applied to the other values that our system rely on such as the DRE of QA practices, execution cost, removal cost, etc.

7.6 Summary

This chapter evaluated our quality management system by simulating data of software development projects. It showed how to utilise our model as a decision support system to make the trade-off process between two or more software QA plans. It outlined how to use our model to calculate the optimal solutions in terms of the execution and escaped cost using the **Linear programming - Simplex Method**. Finally, our methodology for deviation control was proposed which helps the project manager monitor any non-conformity of the current QA activity to the organisation's pre-defined baselines.

Chapter 8

Conclusion & Future Research

8.1 Introduction

This chapter outlines the overall aspects covered in this thesis, the limitations and the assumptions that were not taken into account and highlights the directions of future research. A summary of the thesis chapters is also given in relation to the objectives set out in Chapter 1. The structure of this chapter is as follows. First, Section 8.2 provides a summary of the results obtained through the research presented in this thesis. Then, Section 8.3 presents the conclusions drawn from the results. Finally, Section 8.5 suggests possibilities for improvements to our system and highlights areas of research in the direction of software quality.

8.2 Summary of the Thesis

The ultimate goal of our research was to optimise the investment given to software quality assurance practices in a way that helps make any necessary trade-offs between the software triple constraints: quality, cost and time. In our work presented in this thesis, software quality is looked at as an asset that needs to be well-maintained and saved in order to reduce its negative impact on the overall software schedule and cost.

In the software development process, the cost of software quality (*CoSQ*) consumes a major share of the total software development resources. The reason that drives such high cost is that there is a little consideration given to the fact that optimising investment in QA activities may deliver the level of quality required and at the same time save software resources. Software artifacts should not all have the same consideration in terms of the quality improvement activities due to the fact that there are huge variations in their significance and impact on the system functionality.

There are some software artifacts which hold important architectural components of the system and there are some which are of less significance to the system like the the non-functional attributes of the software. Moreover, shortages in budget assigned to quality improvements or an unexpected cut of the software schedule are common problems that may occur at any interval in the software life cycle.

Based on the foregoing, the project manager needs to consider carefully the priorities and the risk aspects associated with the software under development so as not to appropriately distribute the QA allowance evenly for all modules of the software. In addition, he/she should be able to make informed trade-offs between potential QA plans on the basis of the software triple constraints: quality, cost and schedule. This ability is technically translated in our research into a dynamic decision-based system that works as a support tool for better control and optimisation of the QA investment in software development projects.

This thesis proposed a risk-based approach to software quality investment so that the QA practitioners and software project managers become aware of the potential risk aspects that accompany their software development projects. Our proposed approach incorporates a mechanism process to categorise software modules into work products based on pre-defined risk levels determined according to the organisation scope, software objectives and its prospective domain of use. Based on the categorisation process, the software development organisation will implement their QA activities and store the outcome of these activities in a large repository of QA data. The repository data is channelled and classified according to the risk-based categorised work products for each development phase of the SDLC and for each QA practice ever used within the organisation.

The presented approach should evolve and be applied to many software projects so that the data repository captures a considerable amount of QA data. For new software projects, and after the categorisation process yields a set of new work products the project manager can retrieve experience values from the repository related to each work product. Such values are the estimated defect injection rate, cost of defects escaped from the work product, the efficiency of QA practices ever used with that work product and its relative execution cost and time. Relying on this data, the project manager can get an informed estimates on any potential QA plans which would give a factual approach to decision making.

In addition, the project managers and QA practitioners relying on our model can handle and cope with unforeseen constraints related to their software development process.

They can get optimal QA decisions to deal with budget shortages, schedule reduction or to achieve targets like a target of defect removal success, a minimal quality cost, etc.

8.3 Contribution

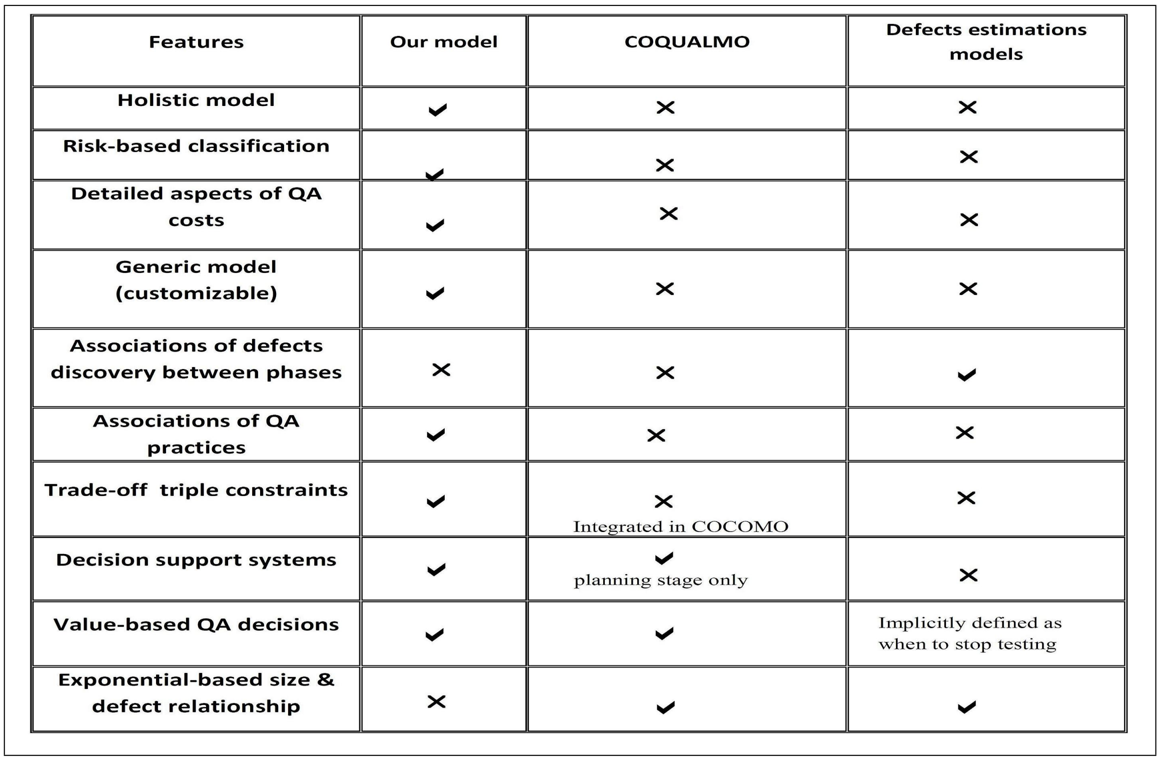
The contributions of our work are twofold. First, an adjustment was proposed to the defect containment matrix, which is currently used in the industry to capture the efficiency of QA activities, to calculate not only the success of an individual QA activity but also the success of the QA practices associated with that activity. The defect containment matrix currently used by QA practitioners measures the success of a QA process by comparing the defects found within this process to the defects escaped as a percentage value named *DRE*. The closer the *DRE* value is to 100%, the more effective the defect detection and removal was for a specific phase or for the whole development life cycle in general.
This DRE value then would be used as a baseline or a metric not only to evaluate the current QA process only but also to evaluate any other similar future software projects. However, this percentage value is general to the whole QA activity without taking into account the practices and techniques used and the differences in defect types and their severities .

In our proposed risk-based approach, the *DRE* value given by our new advanced defect containment matrix takes into account the magnitude and the impact of defects with respect to the work products they originated from. Instead of a single DRE value being generalised for the whole QA process given by the old matrix, the new matrix will quantify defects originating from high risk work products compared with defects that have less of an impact on the software development process. Also, it quantifies the DRE value for each QA practice with respect to the type of work product it applied to so as to output a precise result of the QA activity and the future baselines created.

The other contribution of our research is a QA decision support system that incorporates formal equations to do the following:

- Estimating the number of defects injected into work products or into a whole phase of the software development life cycle.
- Comparing the efficiency and suitability of different QA practices within the software organisation to a specific QA activity.
- Calculating the cost and time of execution and cost of defect removal of QA plan alternatives.
- Generating optimal solutions of QA plans on the basis of the software triple constraints.
- Evaluating QA plans on the basis of the return on investment (ROI) principle.

Our proposed research contributes to the body of knowledge of software engineering by presenting a holistic software QA model to optimise QA practices on the basis of the triple constraints of the software development process: quality, cost and schedule. The project manager with the help of our system can get informed estimate on the consequences of daily-based decisions regarding QA processes. A comparison of our models with similar existing models is depicted in Figure 8.1.



Features	Our model	COQUALMO	Defects estimations models
Holistic model	✓	✗	✗
Risk-based classification	✓	✗	✗
Detailed aspects of QA costs	✓	✗	✗
Generic model (customizable)	✓	✗	✗
Associations of defects discovery between phases	✗	✗	✓
Associations of QA practices	✓	✗	✗
Trade-off triple constraints	✓	✗ Integrated in COCOMO	✗
Decision support systems	✓	✓ planning stage only	✗
Value-based QA decisions	✓	✓	Implicitly defined as when to stop testing
Exponential-based size & defect relationship	✗	✓	✓

Figure 8.1: Comparison with Similar Models

8.4 Limitations

In our research, the following limitations were identified :

- The approach followed in this research deals with a linear-based defect size relationship only without considering the logarithmic relationship. This linearity necessitates that software organisations should have a stable software development process to assure the continuity of the linear relationship and alleviate any factors that may contribute in introducing less or more defects than usual.
- The system's functionality relies entirely on the interaction between each phase within the software development life cycle and the system testing phase. The quantification process of our model's values such as defect removal efficiency, defect

escalation factors, removal cost etc., relies on the association of data resulting from a QA activity in a development phase with the system testing phase.

In order to increase the model accuracy, each phase in the software development life cycle should be interrelated with its successor and predecessor phases as defects injected into a phase may get discovered by the following phase and so on. Accordingly, our model will be used not only to optimise the QA activities within the development phases and the system phase, but also between the phases themselves.

- The proposed model needs to be evaluated using real data from software projects. Having discussed this in the evaluation chapter of our thesis, the application of our system needs to be carried out for several projects within the same organisation in order to build up the data repository of QA activities. This process would take time which exceeds the time allocated for our research. However, plans for future calibration and evaluation of our system are considered and this is going to be pursued after this research is completed.

8.5 Future Work

This section shows how to build on the work presented in this research in a way that improves the efficiency of our system and widens its applicability and prospected domain of use.

In the model of this research and as pointed out in the second point of the limitations section, the interaction is measured between each phase and the system testing phase. Data resulting from this interaction are stored in a large data repository covering all the software development life cycle phases. In future research, interaction between each phase of the software development life cycle needs to be considered and stored in a local repository linked to each phase. The rationale behind this is that some defects which manged to escape from the QA practice applied to a specific work product in phase x may be discovered and removed during the testing activity of phase y. In such cases, values such as the escalation factor ($C^{escaped}$) and *DRE* of those defects and their associated QA practices get more accurate values which in turn have an impact on the accuracy of the decisions made by our system. An overview of the proposed amendment to our model for future research is depicted in Figure 8.2.

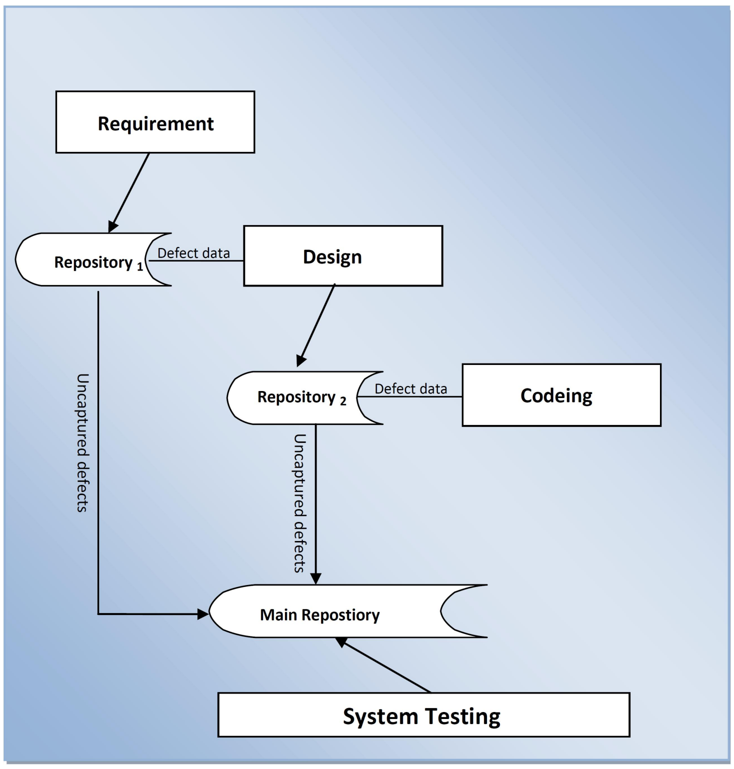


Figure 8.2: Associations Between Phases

This Research is built on the assumption that defects are removed and fixed during the system testing phases if they escaped from the development phases and the QA practice assigned to them. However, in order to make the model more accurate, the cases in which defects are not detected during the system testing phases and escape closer to the delivery to the end customer need to be accounted for. Accordingly, future research may include a risk-based decision support process whereby solutions given by our system to the QA practitioners and to the project managers are normalised by the risk level associated with them. This approach is derived from the ***FMEA***, failure mode and effect analysis, and can be introduced by classifying defects according to a severity level rating on a scale from 1 to 10 . On the other hand, each defect can also be given a detection probability value that will account for the likelihood this defect can be discovered during the system testing phase stage (Figure 8.3).

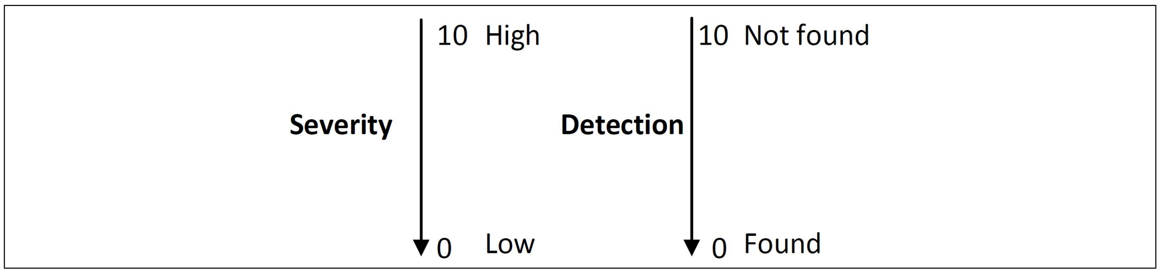
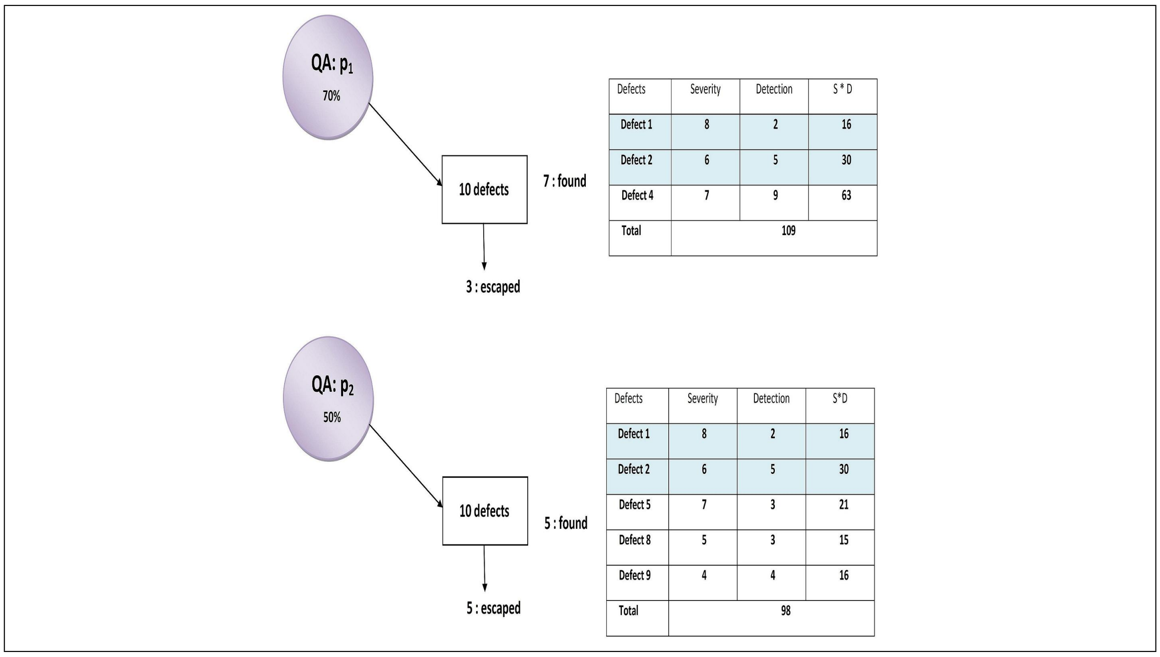


Figure 8.3: Severity and Probability of Detection of Defects

The risk associated with each defect is calculated by multiplying the severity value with the probability of detection and summing up the total.

For example, as shown in Figure 8.4, a QA practice (P_1) which has a defect removal efficiency value DRE = 70% is applied to a work product w_w which is estimated to be injected with 10 defects.

The estimated result of this QA activity is that 7 defects are expected to be found and 3 defects are expected to escape to the system testing phase. On the other hand, another QA practice (P_2) with a DRE of 50% is estimated to find 5 defects and allow 5 defects to propagate to the system testing phase when applied to the same work product. After



Defects	Severity	Detection	S * D
Defect 1	8	2	16
Defect 2	6	5	30
Defect 4	7	9	63
Total	109		

Defects	Severity	Detection	S*D
Defect 1	8	2	16
Defect 2	6	5	30
Defect 5	7	3	21
Defect 8	5	3	15
Defect 9	4	4	16
Total	98		

Figure 8.4: Failure Mode and Effect Analysis

calculating the risk analysis of each of the escaped defects of both P_1 and P_2, the result shows that P_2 has less of a risk implication on the overall QA plan and it may be better to choose it over the first QA practice.

References

[1] R. Adnan, M. Bassem, *A New Software Quality Model for Evaluating COTS Components*, Journal of Computer Science, Vol. 2, No. 4, pp. 373-381, 2006.

[2] M. Agrawal, K. Chari, *Software Effort, Quality, and Cycle Time: A Study of CMM Level 5 Projects*, IEEE Transactions on Software Engineering, March 2007.

[3] F. Akiyama, *An Example of Software System Debuggings*, Proc. Int'l Federation of Information Processing Societies Congress, Vol. 1, pp. 353-359, 1971.

[4] O. Alshathry, J. Helge, Z. Hussein, A. Abdullah, *Quantitative Quality Assurance Approach*,New Trends in Information and Service Science, 2009. NISS'09. International Conference on.

[5] O. Alshathry, H. Janick, *Optimizing Software Quality Assurance*, 2010 34th Annual IEEE Computer Software and Applications Conference Workshops, COMPSAC '10, 2010.

[6] P. Ammann, J. Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008.

[7] N. Ashrafi, *The Impact of Software Process Improvement on Quality: In Theory and Practice*, Information and Management, Vol. 40, No. 7, pp. 677-690, 2003.

[8] A. Aurum, H. Petersson, C. Wohlin, *State-of-the- Art: Software Inspections after 25 Years*, Software Testing, Verification, and Reliability, Vol. 12, No. 3, pp. 133-154, 2002.

[9] A. Aurum, C. Wohlin, *Engineering and Managing Software Requirements*, Springer Verlag, Berlin Germany, p.189, 2005.

[10] B. Basili, B. Perricone, *Software Errors and Complexity: An Empirical Investigation*, Comm. ACM, Vol. 27, No. 1, pp. 42-52, 1984.

[11] A. Bertolino, *Software Testing Research: Achievements, challenges and Dreams*, Future of Software Engineering, L. Briand and A. Wolf (eds.), IEEE-CS Press, 2007.

[12] B. Boehm, C. Abtsand, B. Clark, S. Devnani-Chulani, *COCOMO II Model Definition Manual*, University of Southern California, 1997.

[13] B. Boehm, *Industrial Software Metrics top 10 list*, IEEE Software, September, pages 84-85, 1987.

[14] B. Boehm, V. Basili, *Software Defect Reduction Top 10 List*, IEEE Computer, Vol. 34, No. 1, January 2001.

[15] B. Boehm, R. Madachy, R. Steece, *Software Cost Estimation with COCOMO II*, Prentice Hall, 2000.

[16] B. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.

[17] B. Boehm, L. Huang, *Value-based Software Engineering: A Case Study*, Computer, pages 33-41, March 2003.

[18] B. Boehm, R. Valerdi, *The ROI of Systems Engineering: Some Quantitative Results*, equity,IEEE International Conference on Exploring Quantifiable IT Yields, pp. 79-86, 2007.

[19] L. Briand, K. El Emam, B. Freimut, *Comparison and Integration of Capture-Recapture Models and the Detection Profile Method*, Proc. Ninth International Conference on Software Reliability Engineering, Paderborn, Germany, pp. 32-41, 1998.

[20] N. Bridge, C. Miller, *Orthogonal Defect Classification: Using Defect Data to Improve Software Development*, Software Quality, 1997.

[21] British Standards Institute, http://www.bsi.org.uk.

[22] *Building a Better Bug Trap*, The Economist, June19, 2003, http://www.economist.com/science/tq/displayStory.cfm?Story_id=1841081 [accessed on] 16/oct/2009.

[23] J. Cangussu, S. Haider, K. Cooper, M. Baron, *On the Selection of Software Defect Estimation Techniques*, Software Testing, Verification and Reliability, Wiley Inter-Science, 2009.

[24] C. Catal, B. Diri, *A Systematic Review of Software Fault Prediction Studies*, Expert Systems with Applications, Vol. 36, No. 4, pp. 7346-7354 2009.

[25] C. Ching-Pao, C. Chih-Ping, Y. Yu-Fang, *Estimating the Defect Time Approach for Estimating the Defect Number*, Journal of Software Engineering Studies,Vol. 2, No. 1, pp. 30-43, March 2007.

[26] R. Chillarege, S. Bhandari, K. Chaar, J. Halliday, S. Moebus, K. Ray, Y. Wong, *Orthogonal Defect Classification-A Concept for In- Process Measurements*, IEEE Transactions on Software Engineering, Vol. 18, pp. 943-956, 1992.

[27] M. Ciolkowski, *What Do We Know About Perspective-based Reading? An Approach for Quantitative Aggregation in Software Engineering*, Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 133-144, 2009.

[28] S. Chulani, B. Boehm, *Modeling Software Defect Introduction Removal: CO-QUALMO (COnstructive QUALity MOdel)*, USC-CSE, pp. 99-510, The Center for software Engineering, University of Southern California, Los Angeles, CA, 1999.

[29] B. Chrissis, M. Konard, S. Shrum, *CMMI Guidelines for Process Integration and Product Improvement*, Addison-Wesley, NJ, 2003.

[30] R. Craig, S. Jaskiel, *Systematic Software Testing*, Boston: Artech House, 2002.

[31] B. Crosby, *Quality Is Free: The Art of Making Quality Certain*, McGraw-Hill Book Co., New York (1979).

[32] B. Curtis, W. Hefley, S. Miller, *Overview of the People Capability Maturity Model*. CMU / SEI-95- MM-01. Pittsburgh, 1995.

[33] T. Daughtrey, *Fundamental Concepts for the Software Quality Engineer*, American Society for Quality, 2007.

[34] A. Dennis, W. Barbara, R. Roberta, *System Analysis & design*, Hoboken: John Wiley & Sons, 2006.

[35] M. Dowson, *The ARIANE 501 software failure*, ACM SIGSOFT Software Engineering Notes,2 , 22, pp. 84, March 1997.

[36] A. Dunsmore, M. Roper, M. Wood, *Further Investigations into the Development and Evaluation of Reading Techniques for Object-oriented Code Inspection*, Proceedings of the 24th International Conference on Software Engineering, May 19-25, 2002.

[37] C. Ebert, R. Dumke, M. Bundschuh, A. Schmietendorf, R. Dumke, *Best Practices in Software Measurement*, Springer, New York, 2004.

[38] K. El Emam, A. Birk, *Validating the ISO/IEC 15504 Measure of Software Requirements Analysis Process Capability*, IEEE Transactions on Software Engineering, Vol. 26, No. 6, pp. 541-566, June 2000.

[39] K. El Emam, J. Drouin, W. Melo, *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*, IEEE, Computer Society Press. 1997.

[40] K. El Emam, O. Laitenberger, *Evaluating Capture-Recapture Models with Two Inspectors*, IEEE Trans. Software Eng., Vol. 27, No. 9, pp. 851-864, 2001.

[41] K. El Emam, *The ROI from Software Quality*, Boca Raton, Florida: Auerbach Publications, 2005.

[42] European Space Agency, [accessed on] 16/ Jan/ 2010.

[43] J. Evans, W. Lindsay, *The Management and Control of Quality*, (6th ed.), Mason, OH: Thomson, 2005.

[44] J. Evans, M. Lindsay, *Managing for Quality and Performance Excellence*, (8th ed.), Mason, OH: Thomson Southwestern, 2010.

[45] R. Fairley, M. Willshire, *Iterative Rework: The Good, the Bad, and the Ugly*, IEEE Computer 38 (9): 34-41, 2005.

[46] N. Fenton, N. Ohlsson, *Quantitative Analysis of Faults and Failures in a Complex Software System*, IEEE Transactions on Software Engineering, 2000.

[47] N. Fenton, M. Neil, *A Critique of Software Defect Prediction Research*, IEEE Transactions on Software Engineering, Vol. 25, No. 3, May, 1999.

[48] W. Florac, *Software Quality Measurement: A Framework for Counting Problems and Defects*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, September 1992.

[49] B. Freimut, L. Briand, F. Vollei, *Determining Inspection Cost-Effectiveness by Combining Project Data and Expert Opinion*, IEEE Transactions on Software Engineering, Vol. 31, No. 12, pp. 1074-1092, 2005.

[50] A. Frost, M. Campo, *Advancing Defect Containment to Quantitative Defect Management*, CrossTalk Ű The Journal of Defense Software Engineering 12(20), 24-28, 2007.

[51] Y. Funami, M. Halstead, *A Software Physics Analysis of Akiyama's Debugging Data*, Symposia Series, Computer Software Eng., pp. 133-138, 1976.

[52] R. Futrell, D. Shafer, L. Shafer, *Quality Software Project Management*, Prentice Hall, 2002.

[53] J. Gaffney, *Metrics in Software Quality Assurance*. Proceedings. ACM's 81 Conference, pp. 126-130, 1981.

[54] D. Galin, *Software Quality Assurance, From Theory to Implementation*, Pearson education Ltd., 2004.

[55] D. Garvin, *Competing on the Eight Dimensions of Quality*, Harvard Business Review, pp. 101-109, 1987.

[56] I. Gondra, *Applying Machine Learning to Software Fault-proneness prediction*, Journal of Systems and Software, Vol. 81, No. 2, pp. 186-195, 2008.

[57] L. Gou, Q. Wang, J. Yuan, Y. Yang, M. Li, N. Jiang, *Quantitatively Managing Defects for Iterative Projects: An industrial Experience Report in China*, Heidelberg, D-69121, Germany: Springer Verlag, pp. 369-380, 2008.

[58] L. Gou, Q. Wang, J. Yuan, Y. Yang, M. Li, N. Jiang, *Quantitative Defects Management in Iterative Development with BiDefect*, Software Process: Improvement and Practice, Vol. 14, No. 4, pp. 227-241, 2008.

[59] T. Gyimothy, R. Ferenc, I. Siket, *Empirical Validation of Object-oriented Metrics on Open Source Software for Fault Prediction*, IEEE Transactions on Software Engineering, Vol. 31, No. 10, pp. 897-910, 2005.

[60] M. Halstead, *Elements of Software Science*, Elsevier, 1977

[61] V. Hans, *Software Engineering: Principles and Practice*, 2nd Edition, Wiley, Sept, 2000.

[62] D. Harter, S. Krishnan, A. Slaughter,*Effects of Process Maturity on Quality, Cycle Time, and Effort in Software Product Development*, Management Science, Vol. 46, No. 4, pp. 451-466, April 2000.

[63] C. Henderson, *Managing Software Defects: Defect Analysis and Traceability*, ACM SIGSOFT Software Engineering Notes, 2008.

[64] D. Houston, J. Keats, *Cost of Software Quality: A Means of Promoting Software Process Improvement*, Quality Engineering, Vol.10, No. 3, pp. 563-573, March, 1998.

[65] S. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley Publishing Company,Massachusetts, 1995.

[66] B. Huitfeldt, M. Middleton, *The Assessment of Software Quality From the User Perspective: Evaluation of a GIS implementation*, Journal of End User Computing, Vol. 13, No. 1, pp. 3-11, 2001.

[67] R. Hunter, R. Thayer, *Software Process Improvement*, IEEE Computer Society, Los Alamitos, California, 2001.

[68] IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology, 1990.

[69] J. Ingalsbe, D. Shoemaker, V. Jovanovic, *A Meta Model for the Capability Maturity Model for Software*, AMCIS 2001 Proceedings. pp. 253-259, 2001.

[70] S. Isoda, *A Criticism on the Capture and Recapture Method for Software Reliability Assurance*, Journal of Systems and Software, 43, pp. 3-10, 1998.

[71] ISO 9001 Quality, http://www.bsi-global.com/en/Assessment-and-certification-services/management-systems/Standards-and-Schemes/ISO-9001.

[72] S. Jaju, R. Mohanty, R. Lakhe, *Towards Managing Quality Cost: A Case Study*, Total Quality Management & Business Excellence, pp. 1 075 - 1094, 2009.

[73] L. Jeremy, *SDLC 100 Success Secrets - Software Development Life Cycle (SDLC) 100 Most asked Questions, SDLC Methodologies, Tools, Process and Business Models*, Emereo Pty Ltd, 2008.

[74] H. Jia, F. Shu, Y. Yang, Q. Wang, *Predicting Fault-Prone Modules: A Comparative Study*, Springer Berlin Heidelberg, 2009.

[75] Z. Jiang, P. Naude, B. Jiang, *The Effects of Software Size on Development Effort and Software Quality*, Proc. World Academy Sci, Eng and Tech 23, pp. 363-367, 2007.

[76] J. Johnson, *My Life Is Failure: 100 Things You Should Know to be a Successful Project Leader*, Standish Group International, West Yarmouth, MA, 2006.

[77] C. Jones, *Applied Software Measurement: Global Analysis of Productivity and Quality*, 3rd edition, New York : McGraw-Hill, 2008.

[78] C. Jones, *Estimating Software Costs: Bringing Realism to Estimation*, 2nd edition. McGraw-Hill, New York, 2007.

[79] C. Jones, *The Economics of Software Process Improvement*, IEEE Computer, Vol. 29, pp. 95-97, 1996.

[80] C. Jones, *Programming Defect Removal*, Proceedings, Vol. 40, GUIDE, USA, 1975.

[81] Z. Jun, *Cost-sensitive Boosting Neural Networks for Software Defect Prediction*, Expert Systems with Applications, Vol. 37, pp. 4537-4543, 2010.

[82] N. Juristo, A. Moreno, S. Vegas, *Reviewing 25 Years of Testing Technique Experiments*, Empirical Software Engineering, Vol. 9, pp. 7-44, 2004.

[83] J. Juran, *How to Think About Quality*, Juran's quality handbook, 5th ed., New York: McGraw-Hill, pp. 2.1-2.3, 1998.

[84] S. Kan, *Metrics and Models in Software Quality Engineering*, Addison Wesley, 2nd edition, 2002.

[85] A. Koru, H. Liu, *Building Effective Defect Prediction Models in Practice*, IEEE Software, pp. 23-29, 2005.

[86] A. Koru, J. Tian, *An Empirical Comparison and Characterization of High Defect and High Complexity Modules*, Journal of Systems and Software 67, pp. 153-163, 2003.

[87] G. Koru, D. Zhang, K. El Emam, H. Liu, *An Investigation into the Functional Form of the Size-Defect Relationship for Software Modules*, IEEE Transactions on Software Engineering, Vol. 35, No. 2, pp. 293-304, 2009.

[88] H. Krasner, *Using the Cost of Quality Approach for Software*, CrossTalk. The Journal of Defense Software Engineering, Vol. 11, No. 11, pp. 6-11, 1998.

[89] L. Lazic, N. Mastorakis, *Cost Effective Software Test Metrics*, WSEAS Transactions on Computer, 6, 7, pp. 599-619, 2008.

[90] O. Laitenberger, *A Survey of Software Inspection Technologies*, Handbook on Software Engineering and Knowledge Engineering, Vol. II, 2002.

[91] O. Laitenberger, J. DeBaud, *An Encompassing Life-Cycle Centric Survey of Software Inspection*, International Software Engineering Research Network (ISERN) Technical Report ISERN-98-14, Fraunhofer Institute for Experimental Software Engineering, 1997.

[92] L. Lazic, A. Kolasinac, D. Avdic, *The Software Quality Economics Model for Software Project Optimization*, World Scientific and Engineering Academy and Society (WSEAS), Vol. 8, No. 1, January 2009.

[93] A. Lattanze, *Architecting Software Intensive Systems: A Practitioners Guide*, Auerbach Publications, 2008.

[94] H. Liguo, B. Boehm, *How Much Software Quality Investment Is Enough: A Value-Based Approach*, IEEE software, pp.88-95, 2006.

[95] M. Lipow, *Number of Faults per Line of Code*, IEEE Trans. Software Eng., Vol. 8, No. 4, pp. 437-439, 1982.

[96] A. MacCormack, *Product-Development Processes That Work: How Internet Companies Build Software*, Sloan Management Review, Vol. 42, No. 1, pp.75-84, 2001.

[97] F. Machowski, G. Dale,*Quality Costing: An Examination of Knowledge, Attitudes and Perceptions*, Quality Management Journal, Vol. 5, No. 3, p. 84, 1998.

[98] R. Madachy, B. Boehm, *Assessing Quality Processes with ODC COQUALMO*, In Wang, Q., Pfahl, D., Raffo, D. (eds.) Making Globally Distributed Software Development a Success Story, pp. 198-209 Springer-Verlag, Berlin, 2008.

[99] B. Manfred, *Editorial-Science of Computer Programming-25 years*, Science of Computer Programming, Vol. 66, No. 2, 30 April 2007.

[100] T. McCabe, *SQA-A Survey, Columbia*, OH: McCabe Press, pp. 154-156, 1980.

[101] J. McCall, P. Richards, G. Walters, *Factors in Software Quality*, General Electric, National Technical Information Service, 1977.

[102] S. McConnell, *Code Complete*, 2nd Ed., Microsoft Press, 2004.

[103] K. Moller, D. Paulish, *An Empirical Investigation of Software Fault Distribution*, Proc. First IntŠl Software Metrics Symp., pp. 82- 90, May 1993.

[104] G. Myers, *The Art of Software Testing*, John Wiley & Sons, second edition, 2004.

[105] N. Nagappan, *Static Analysis Tools as Early Indicators of Pre-Release Defect Density*, Proceedings of International Conference on Software Engineering, pp. 580-586, 2005.

[106] N. Nagappan, B. Murphy, V. Basili, *The Influence of Organizational Structure on Software Quality: an Empirical Case Study*, In ICSE '08: Proceedings of the 30th international conference on Software engineering, pages 521Ű530, New York, NY, USA, 2008. ACM.

[107] S. Naik, P. Tripathy, *Software Testing and Quality Assurance*, John Wiley & Sons, 2008.

[108] *NASA Procedures and Guidelines for Mishap Reporting, Investigating and Record keeping*, Safety and Risk management Division, NASA Headquarters, USA, 2000.

[109] NIST, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, National Institute of Standards and Technology, U.S. Department of Commerce, 2002.

[110] A. Olalekan, O. Adenike, *An Empirical Comparative Study of Checklist-based and Ad Hoc Code Reading Techniques in a Distributed Groupware Environment*, (IJCSIS) International Journal of Computer Science and Information Security, Vol. 5, No. 1, 2009.

[111] T. Ostrand, E. Weyuker, *The Distribution of Faults in a Large Industrial Software System*, In Proceedings of the International Symposium on Software Testing and Analysis, Rome, 2002.

[112] G. O'Regan, *A Practical Approach to Software Quality*, 1st edition, Springer, 2002.

[113] M. Osamu, H. Hata, *An Empirical Comparison of Fault-prone Module Detection Approaches:Complexity Metrics and Text Feature Metrics*, 2010 IEEE 34th Annual Computer Software and Applications Conference, 2010.

[114] L. Osterweil, *Strategic Directions in Software Quality*, ACM Computing Surveys, Vol. 28, No. 4, pp. 738-750, 1996.

[115] T. Pan, L. Zheng, C. Fang, *Defect Tracing System Based on Orthogonal Defect Classification*, International Conference on Computer Science and Software Engineering, 2008.

[116] M. Paulk, C. Weber, W. Curtis, M. Chrissis, *The Capability Maturity Model: Guidelines for Improving the Software Process*. Carnegie Mellon University Software Engineering Institute, 1994.

[117] M. Paulk, *Agile Methodologies and Process Discipline*, http://www.stsc.hill.af.mil/CrossTalk/2002/oct/paulk.asp, CrossTalk, 2002.

[118] J. Pai, B. Dugan, *Empirical Analysis of Software Fault Content and Fault Proneness Using Bayesian Method*, IEEE Transactions on Software Engineering, Vol. 33, No. 10, pp. 675-686, 2007.

[119] H. Peng, Z. Jie, *Predicting Defect-Prone Software Modules at Different Logical Levels*, International Conference on Research Challenges in Computer Science, 2009.

[120] W. Perry, *Effective Methods for Software Testing*, 3rd edition, John Wiley & Sons: New York, NY, 2006.

[121] S. Pfleeger, *Does Organizational Maturity Improve Quality ?*, IEEE Software, Vol. 13, No. 15, pp. 109-110, 1996.

[122] J. Plunkett, B. Dale, *A Review of the Literature on Quality-related Costs*, International Journal of Quality and Reliability Management, Vol. 4, No. 1, pp. 40, 1987.

[123] Price Waterhouse, *Software Quality Standards: The Costs and Benefits*, A Review for the Department of Trade and Industry. London: Price Waterhouse Management Consultants, 1988.

[124] A. Porter, L. Votta, V. Basili, *Comparing Detection Methods for Software Requirements Inspections: A replicated experiment*, IEEE Trans. on Software Engineering, 21 Harvey, pp. 563-575, 1996.

[125] Z. Qingyu, *Quality Dimensions, Perspectives and Practices*, International Journal of Quality & Reliability Management, Vol. 18, pp. 708-722, 2001.

[126] B. Robert, *Practical Software Metrics for Project Management and Process Improvement*, Englewood Cliffs, NJ: Prentice Hall, 1992.

[127] B. Robert, L. Phillip, *A Dynamic Capture-recapture Model for Software Defect Prediction*, Innovations System software engineering, pp. 265-270, 2009.

[128] S. Robertson, R. Robertson, *Mastering the Requirements Process*, Harlow, UK, 2006.

[129] J. Rodgers, W. Nicewander, *Thirteen Ways to Look at the Correlation Coefficient*, The American Statistician, 42, pp:59.66, 1988.

[130] A. Sandoval-Chavez, M. Beruvides, *Using Opportunity Costs to Determine the Cost of Quality: A Case Study in a Continuous-Process Industry*, The Engineering Economist; Vol. 43, No. 2, pp. 107-124, 1998.

[131] A. Schiffauerova, V. Thomson, *A Review of Research on Cost of Quality Models and best practices*, International Journal of Quality & Reliability Management, Vol. 23, No. 6, pp. 647-659, 2006.

[132] R. Seider, *Implementing Phase Containment Effectiveness Metrics at Motorola*, Crosstalk, 2006.

[133] A. Slaughter, D. Harter, S. Drishnan, *Evaluating the Cost of Software Quality*, In Communications of the ACM, Vol. 41, pp. 67-73, 1998.

[134] V. Sower, R. Quarles, S. Cooper, *Cost of Quality:Distribution and Quality System Maturity: An Exploratory Study*, ASQ's 56th Annual Quality Congress Proceedings, ASQ, May, 2002.

[135] *Software Process Improvement and Capability dEtermination*, http://www.sqi.gu.edu.au/spice/, [accessed on] 19/ Dec/ 2009.

[136] C. Stringfellow, A. Andrews, *An Empirical Method for Selecting Software Reliability Growth Models*, Empirical Software Engineering, 7(4): pp. 319-343, 2002.

[137] *Standish Group International. The Chaos Report*, 2004 www.standishgroup.com/sample_ research/PDFpages/Chaos1994.pdf.

[138] V. Suma, T. Gopalakrishnan, *Effective Defect Prevention Approach in Software Process for Achieving Better Quality Levels*, World Academy of Science, Engineering and Technology, 42, 2008.

[139] T. Thelin, P. Runeson, C. Wohlin, *An Experimental Comparison of Usage-Based and Checklist-Based Reading*, IEEE Transactions on Software Engineering, Vol. 29, No. 8, pp. 687-704, 2003.

[140] T. Thelin, C. Andersson, P. Runeson, N. Dzamashvili- Fogelstrm, *A Replicated Experiment of Usage-Based and Checklist-Based Reading*, Proc. of 10th Int. Symp. on Software Metrics, 2004.

[141] T. Thelin, P. Runeson, B. Regnell, *Usage-Based Reading - An Experiment to Guide Reviewers with Use Cases*, Information and Software Technology, Vol. 43, No. 15, pp. 925-938, 2001.

[142] J. Tian, *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable improvement*, Wiley, pp. 15-25, 2005.

[143] W. Tsai, *Quality Cost Measurement Under Activity-Based Costing*, International Journal of Quality & Reliability Management, Vol. 15, pp. 719-752, 1998.

[144] B. Unhelkar, *Practical Object Oriented Analysis*, Thomson, 2005.

[145] L. Votta, *Does Every Inspection Need a Meeting ?*, ACM Software Eng., Vol. 18, No. 5, pp. 107-114, 1993.

[146] G. Walia, J. Carver, *Evaluation of Capture-recapture Models for Estimating the Abundance of Naturally-occurring Defects*, Proc. of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, pp. 158-167, 2008.

[147] S. Wanger, *A Literature Survey of the Quality Economics of Defect Detection Techniques*, Presented at International Symposium on Empirical Software Engineering (ISESE '06), ACM Pres, pp. 194-203, 2003.

[148] Q. Wang, L. Gou, N. Jiang, M. Che, R. Zhang, Y. Yang, M. Li, *An Empirical Study on Establishing Quantitative Management Model for Testing Process*, JCSP 2007, Lecture Notes on Computer Science, Vol. 4470, 2007.

[149] E. Weyuker, *Comparing the Effectiveness of Testing Techniques*, Formal Methods and Testing, pp. 271-291, 2008.

[150] B. Whitehall, *Reviews of Problems with a Quality Costing System*, International Journal of Quality & Reliability Management, Vol. 3, pp. 43-58, 1986.

[151] C. Wohlin, P. Runeson, *Defect Content Estimations from Review Data*, Procs. International Conference on Software Engineering, Kyoto, Japan, pp. 400 - 409, 1998.

[152] M. Wood, M. Roper, A. Brooksand, J. Miller, *Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study*, Proceedings of the 6th European Software Engineering Conference, Zurich, Switzerland, pp. 262-277, 1997.

[153] A. Wood, *Software Reliability Growth Models: Assumptions vs. Reality*, In ISSRE '97: Proc. of the 8th IEEE International Symposium on Software Reliability Engineering, Los Alamitos,IEEE Computer Society, CA, USA, 1997.

[154] Y. Wong, *Modern Software Review: Techniques and Technologies*, IRM Press, 2006.

[155] H. Zhang, X. Zhang, M. Gu, *Predicting Defective Software Components from Code Complexity Measures*, in Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing, Melbourne, Australia, pp. 93-96, 2007.

[156] K. Zou, K. Tuncali, S. Silverman, *Correlation and Simple Linear Regression*, Radiology, 227, pp. 617-622, 2003.

Appendices

Appendix A

Source Code of the Main Classes

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Data.OleDb;

namespace QA
{
    public partial class Main : Form
    {
        public Main()
        {
            InitializeComponent();
            ShowDate();
```

```
}

private void button1_Click(object sender, EventArgs e)
{
    OleDbConnection con = new OleDbConnection();
    OleDbCommand cmd = new OleDbCommand();

    try
    {
        if (txtTechnique.Text.Length > 0)
        {
            if (label1.Text.Length > 0)
            {
                con.ConnectionString = fnConnectionString();
                con.Open();
                cmd.Connection = con;
                int intValue = 0;

                intValue =int.Parse(label1.Text);
                cmd.CommandText = "update [Technique$] set technique
                     ='" + txtTechnique.Text.Trim() + "' where sno= "
                     + intValue;
                cmd.ExecuteNonQuery();
                con.Close();
                btnTechnique.Text = "Add";
            label1.Text = "";
        }
        else
        {


            con.ConnectionString = fnConnectionString();
            con.Open();
            cmd.Connection = con;
            int intValue = 0;
            cmd.CommandText = "select count(sno) from [
                Technique$]";
            intValue = int.Parse(cmd.ExecuteScalar().ToString())
                ;
            if (intValue == 0)
            {


            }
            else
            {
            cmd.CommandText = "select max(sno) from [Technique$]
                ";
            intValue = int.Parse(cmd.ExecuteScalar().ToString())
                ;
            }
```

```
                    intValue = intValue + 1;

                    cmd.CommandText = "insert into [Technique$](sno,

                       technique,isDeleted) values ('" + intValue + "','

                       " + txtTechnique.Text.Trim() + "',false)";

                    cmd.ExecuteNonQuery();

                    con.Close();

                 }

              ShowDate();

              txtTechnique.Text = "";

              }

              else

              {

                 MessageBox.Show("Enter Technique");


              }







           }

           catch(Exception ex)

           {

              MessageBox.Show(ex.Message);

              con.Close();

           }


        }
```

```
private string fnConnectionString ()
{
    string strConnection;
    strConnection = Properties.Settings.Default.cString;
    string strPath;
    strPath = System.IO.Directory.GetCurrentDirectory();
    strPath = strPath + "\\" + strConnection;
    strConnection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source='
        " + strPath + "';Extended Properties='Excel 8.0;HDR=YES;'";

    return strConnection;
}
private void ShowDate()
{

    OleDbConnection con = new OleDbConnection();
    OleDbCommand cmd = new OleDbCommand();
    DataSet ds = new DataSet();
    con.ConnectionString =fnConnectionString();
    cmd.Connection =con;
    cmd.CommandText = "select sno,technique from [Technique$] where
         isDeleted=false ";
    try
    {
        OleDbDataAdapter dbAdp = new OleDbDataAdapter(cmd);
        //dbAdp.SelectCommand = "select * from [Technique$]";
```

```
            dbAdp.Fill(ds);

            dgTechniqueEntry.DataSource = ds.Tables[0] ;


            dbAdp.Dispose();





        }

        catch (Exception ex)

            {

            MessageBox.Show (ex.Message );

            con.Close();

            }




    }


    private void button1_Click_1(object sender, EventArgs e)

    {

        ShowDate();

    }




    private void dgTechniqueEntry_CellClick(object sender,

        DataGridViewCellEventArgs e)

    {
        if (e.ColumnIndex == 1)
        {
            int strValue = 0;
            strValue = e.RowIndex;
            //?dgTechniqueEntry.Rows[e.RowIndex ].Cells[0].Value
            txtTechnique.Text = dgTechniqueEntry.Rows[e.RowIndex].Cells
                [1].Value.ToString() ;
            btnTechnique.Text = "Update";
            label1.Text = dgTechniqueEntry.Rows[e.RowIndex].Cells[0].
                Value.ToString();
            label1.Visible = false;
            btnDelete.Visible = true;

        }
    }

    private void txtTechnique_Validating(object sender, CancelEventArgs
         e)
    {
        string strError="";
        if (txtTechnique.Text.Length ==0 )
        {
            strError="Enter project name";
            e.Cancel = true;
        }
        errorTechnique.SetError((Control)sender, strError);
    }

    private void btnDelete_Click(object sender, EventArgs e)
    {
        OleDbConnection con = new OleDbConnection();
        OleDbCommand cmd = new OleDbCommand();

        try
        {
            if (txtTechnique.Text.Length > 0)
            {
                if (label1.Text.Length > 0)
                {
                    con.ConnectionString = fnConnectionString();
                    con.Open();
                    cmd.Connection = con;
                    int intValue = 0;

                    intValue = int.Parse(label1.Text);

                    cmd.CommandText = "update [Technique$] set isdeleted
                        =true where sno= " + intValue;
                    cmd.ExecuteNonQuery();
                    cmd.CommandText = "update [BugEntry$] set isdeleted=
                        true where techSno= " + intValue;
                    cmd.ExecuteNonQuery();
```

```
                        con.Close();
                        btnDelete.Visible = false;
                        btnLatestData.Text = "Add";
                        label1.Text = "";
                        MessageBox.Show("Record Deleted Successfully");
                    }

                    ShowDate();
                    txtTechnique.Text = "";

                }
                else
                {
                    MessageBox.Show("Enter Technique");

                }




            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
                con.Close();
            }
        }
```

```
    }
}
```

Listing A.1: Source Code of Main Class

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Data.OleDb;

namespace QA
{
    public partial class CostTime : Form
    {
        public CostTime()
        {
```

```
        InitializeComponent();
    }

    private void btnShow_Click(object sender, EventArgs e)
    {
        OleDbConnection con = new OleDbConnection();
        OleDbCommand cmd = new OleDbCommand();
        DataSet ds = new DataSet();
        try
        {



            con.ConnectionString = fnConnectionString();
            con.Open();
            cmd.Connection = con;

            if (cmbType.SelectedItem.ToString() != "Select")
            {
                //cmd.CommandText = "select sum(perFound) as FSum , sum(
                    perEscape) as ESum, sum(Dre) as TDre, Count(
                    fountDefect) as CCount from [BugEntry$] where
                    isdeleted=false and projSno =" + cmbProject.
                    SelectedValue.ToString() + " and techSno=" +
                    cmbTechnique.SelectedValue.ToString() + " and type
                    ='" + cmbType.SelectedItem + "'";
                //cmd.CommandText = "select sum(removalcost) as FSum ,
        sum(Executontime) as ESum, Count(removalcost) as
        CCount from [BugEntry$] where isdeleted=false and
        projSno =" + cmbProject.SelectedValue.ToString() + "
         and techSno=" + cmbTechnique.SelectedValue.ToString
        () + " and type='" + cmbType.SelectedItem + "'";
    cmd.CommandText = "select sum(removalcost) as FSum , sum
        (Executontime) as ESum, Count(removalcost) as CCount
         from [BugEntry$] where isdeleted=false and techSno=
        " + cmbTechnique.SelectedValue.ToString() + " and
        type='" + cmbType.SelectedItem + "'";
    OleDbDataAdapter dbAdp = new OleDbDataAdapter(cmd);
    dbAdp.Fill(ds);

    float fvalue1 = 0, fvalue2 = 0, fresult = 0;
    if (ds.Tables[0].Rows[0][0].ToString().Trim() != "")
    {
        fvalue1 = float.Parse(ds.Tables[0].Rows[0][0].
            ToString().Trim());
        fvalue2 = float.Parse(ds.Tables[0].Rows[0][2].
            ToString().Trim());
        fresult = (fvalue1 / fvalue2);
        txtAvgRemovalCost.Text = Convert.ToString(Decimal.
            Round(decimal.Parse(fresult.ToString()), 2));
    }
    else
    {
```

```
                txtAvgRemovalCost.Text = "0";
            }
            if (ds.Tables[0].Rows[0][1].ToString().Trim() != "")
            {
                fvalue1 = float.Parse(ds.Tables[0].Rows[0][1].
                    ToString().Trim());
                fvalue2 = float.Parse(ds.Tables[0].Rows[0][2].
                    ToString().Trim());
                fresult = (fvalue1 / fvalue2);
                txtExecutionTime.Text = Convert.ToString(Decimal.
                    Round(decimal.Parse(fresult.ToString()), 2));
            }
            else
            {
                txtExecutionTime.Text = "0";
            }



        }
        else
        {
            MessageBox.Show("Please Select the Details");
        }

    }
```

```
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
                con.Close();
            }
        }
        private string fnConnectionString()
        {
            string strConnection;
            strConnection = Properties.Settings.Default.cString;
            string strPath;
            strPath = System.IO.Directory.GetCurrentDirectory();
            strPath = strPath + "\\"+strConnection;
            strConnection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source='
                " + strPath + "';Extended Properties='Excel 8.0;HDR=YES;'";

            return strConnection;
        }
        private void ShowDate()
        {


            OleDbConnection con = new OleDbConnection();
            OleDbCommand cmd = new OleDbCommand();
            DataSet ds = new DataSet();
            DataSet ds2 = new DataSet();
            con.ConnectionString = fnConnectionString();
```

```
cmd.Connection = con;
cmd.CommandText = "select sno,technique from [Technique$] where
    isDeleted=false ";
try
{
    OleDbDataAdapter dbAdp = new OleDbDataAdapter(cmd);
    dbAdp.Fill(ds);

    cmbTechnique.DataSource = ds.Tables[0];
    cmbTechnique.DisplayMember = "technique";
    cmbTechnique.ValueMember = "sno";
    // cmbTechnique.Items.Insert(0, "Select");
    //cmbTechnique.Items.Insert(0,string.Empty );
    //cmbTechWeight.SelectedIndex =-1;
    dbAdp.Dispose();

    ////cmd.CommandText = "select sno,Projectname from [
        ProjectDetail$] where isDeleted=false ";

    ////OleDbDataAdapter dbAdpSecond = new OleDbDataAdapter(cmd
        );
    ////dbAdpSecond.Fill(ds2);

    ////cmbProject.DataSource = ds2.Tables[0];
    ////cmbProject.DisplayMember = "Projectname";
    ////cmbProject.ValueMember = "sno";
```

```
        //////cmbProject.Items.Insert(0, "Select");
        //////cmbTechWeight.SelectedIndex = 0;

        ////dbAdpSecond.Dispose();

        cmbType.Items.Add("High");
        cmbType.Items.Add("Medium");
        cmbType.Items.Add("Low");
        cmbType.Items.Insert(0, "Select");
        cmbType.SelectedIndex = 0;









    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
        con.Close();
    }
```

```
        }


        private void CostTime_Load(object sender, EventArgs e)
        {
            ShowDate();
        }
    }
}
```

Listing A.2: Source Code of CostTime Class

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Data.OleDb;


namespace QA
{
    public partial class BugEntry : Form
    {
```

```
public BugEntry()
{
    InitializeComponent();


}


private void BugEntry_Load(object sender, EventArgs e)
{
    ShowDate();
   // cmbProject_SelectedIndexChanged(null, null);
}
private string fnConnectionString()
{
    string strConnection;
    strConnection = Properties.Settings.Default.cString;
    string strPath;
    strPath = System.IO.Directory.GetCurrentDirectory();
    strPath = strPath + "\\" + strConnection;
    strConnection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source='
        " + strPath + "';Extended Properties='Excel 8.0;HDR=YES;'";

    return strConnection;
}
private void ShowDate()
{
        OleDbConnection con = new OleDbConnection();

        OleDbCommand cmd = new OleDbCommand();

        DataSet ds = new DataSet();

        DataSet ds2 = new DataSet();

        con.ConnectionString = fnConnectionString();

        cmd.Connection = con;

        cmd.CommandText = "select sno,technique from [Technique$] where
             isdeleted=false";

        try

        {

            OleDbDataAdapter dbAdp = new OleDbDataAdapter(cmd);

            dbAdp.Fill(ds);


            cmbTechnique.DataSource = ds.Tables[0];

            cmbTechnique.DisplayMember = "technique";

            cmbTechnique.ValueMember = "sno";

            // cmbTechnique.Items.Insert(0, "Select");

            //cmbTechnique.Items.Insert(0,string.Empty );

            //cmbTechWeight.SelectedIndex =-1;

            dbAdp.Dispose();


            cmd.CommandText = "select sno,Projectname from [
                ProjectDetail$] where isdeleted=false";


            OleDbDataAdapter dbAdpSecond = new OleDbDataAdapter(cmd);

            dbAdpSecond.Fill(ds2);
            cmbProject.DataSource = ds2.Tables[0];
            cmbProject.DisplayMember = "Projectname";
            cmbProject.ValueMember = "sno";

            //cmbProject.Items.Insert(0, "Select");
            //cmbTechWeight.SelectedIndex = 0;

            dbAdpSecond.Dispose();

            cmbTechWeight.Items.Add("High");
            cmbTechWeight.Items.Add("Medium");
            cmbTechWeight.Items.Add("Low");
            cmbTechWeight.Items.Insert(0, "Select");
            cmbTechWeight.SelectedIndex = 0;









        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
```

```
            con.Close();
        }



    }

    private void btnAddBug_Click(object sender, EventArgs e)
    {
        OleDbConnection con = new OleDbConnection();
        OleDbCommand cmd = new OleDbCommand();

        try
        {
            //txtPercEscapeDefect.Text = "0";
            //txtPerFoundDefect.Text = "0";
            //txtSizeofProduct.Text = "0";

            //txtExecutionCost.Text = "0";
            //txtRemovalCost.Text = "0";
            string strMessage = "";
            if (cmbTechWeight.SelectedIndex > 0)
            {
                txtWeight_TextChanged(null, null);
                txtDefect_TextChanged(null, null);
                txtEscapeDefect_TextChanged(null, null);
                txtCost_TextChanged(null, null);
            txtTime_TextChanged(null, null);
            txtPage_TextChanged(null, null);
            if (lblHiddenValue.Text.Length > 0)
            {
                con.ConnectionString = fnConnectionString();
                con.Open();
                cmd.Connection = con;
                int intValue = 0;

                intValue = int.Parse(lblHiddenValue.Text);
                cmd.CommandText = "update [BugEntry$] set projSno ='
                    " + cmbProject.SelectedValue.ToString() + "',
                    techSno ='" + cmbTechnique.SelectedValue.ToString
                    () + "',type ='" + cmbTechWeight.SelectedItem.
                    ToString() + "',techWeight= '" + txtWeight.Text.
                    Trim() + "',sizeworkProduct ='" +
                    txtSizeofProduct.Text.Trim() + "',FoundDefect ='"
                     + txtDefect.Text.Trim() + "',perFound ='" +
                    txtPerFoundDefect.Text.Trim() + "',EscapeDefect
                    ='" + txtEscapeDefect.Text.Trim() + "',perEscape
                    ='" + txtPercEscapeDefect.Text.Trim() + "',page
                    ='" + txtPage.Text.Trim() + "',cost ='" + txtCost
                    .Text.Trim() + "',timeduration ='" + txtTime.Text
                    .Trim() + "',removalcost ='" + txtRemovalCost.
                    Text.Trim() + "',Executontime ='" +
                    txtExecutionCost.Text.Trim() + "',dre='"+txtDRE.
                Text.Trim() +"’ where BugSno= " + intValue;
            cmd.ExecuteNonQuery();
            con.Close();
            btnAddBug.Text = "Add";
            lblHiddenValue.Text = "";
            strMessage = "Record Updated Successfully";
        }
        else
        {


            con.ConnectionString = fnConnectionString();
            con.Open();
            cmd.Connection = con;
            int intValue = 0;
            cmd.CommandText = "select count(BugSno) from [
                BugEntry$]";
            intValue = int.Parse(cmd.ExecuteScalar().ToString())
                ;
            if (intValue == 0)
            {


            }
            else
            {
                cmd.CommandText = "select max(BugSno) from [
                    BugEntry$]";
                intValue = int.Parse(cmd.ExecuteScalar().ToString
                    ());
            }
            intValue = intValue + 1;
            strMessage = "insert into [BugEntry$](BugSno,projSno
                 ,techSno,type,techWeight,sizeworkProduct,
                FoundDefect,perFound,EscapeDefect,perEscape,dre,
                page,cost,timeduration,removalcost,Executontime,
                isDeleted) values ";
            strMessage = strMessage + "('" + intValue + "','" +
                cmbProject.SelectedValue.ToString() + "','" +
                cmbTechnique.SelectedValue.ToString() + "','" +
                cmbTechWeight.SelectedItem.ToString() + "','" +
                txtWeight.Text.Trim() + "','" + txtSizeofProduct.
                Text.Trim() + "','" + txtDefect.Text.Trim() + "
                ','" + txtPerFoundDefect.Text.Trim() + "','" +
                txtEscapeDefect.Text.Trim() + "','" +
                txtPercEscapeDefect.Text.Trim() + "','"+txtDRE.
                Text.Trim() +"','" + txtPage.Text.Trim() + "','"
                + txtCost.Text.Trim() + "','" + txtTime.Text.Trim
                () + "','" + txtRemovalCost.Text.Trim() + "','" +
                 txtExecutionCost.Text.Trim() + "',false)";
            cmd.CommandText = strMessage;
            cmd.ExecuteNonQuery();
            con.Close();
            strMessage = "Record Added Successfully";
            }
            //ShowDate();
            txtDefect.Text = "0";
            txtEscapeDefect .Text = "0";
            txtPercEscapeDefect.Text = "0";
            txtPerFoundDefect .Text = "0";
            txtSizeofProduct .Text = "0";
            txtWeight.Text = "0";

            txtDRE.Text = "0";
            txtPage.Text = "0";
            txtCost.Text = "0";
            txtTime.Text = "0";
            txtExecutionCost.Text = "0";
            txtRemovalCost.Text = "0";

            btnProjectDetails_Click(null, null);
            MessageBox.Show(strMessage );
            btnDelete.Visible = false;

        }
        else
        {
            MessageBox.Show("Enter details");

        }
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
            con.Close();
        }
    }

    //private void cmbProject_SelectedIndexChanged(object sender,
        EventArgs e)
    //{
    //    if (cmbProject.SelectedIndex > 0)
    //    {
    //        OleDbConnection con = new OleDbConnection();
    //        OleDbCommand cmd = new OleDbCommand();
    //        DataSet ds = new DataSet();
    //        con.ConnectionString = fnConnectionString();
    //        cmd.Connection = con;
    //        cmd.CommandText = "select * from [BugEntry$] where projSno
        ="+cmbProject.SelectedValue+"" ;
    //        try
    //        {
    //            OleDbDataAdapter dbAdp = new OleDbDataAdapter(cmd);
```

```
//            dbAdp.Fill(ds);
//            dgBugEntry.DataSource = ds.Tables[0];

//            dbAdp.Dispose();




//        }
//        catch (Exception ex)
//        {
//            MessageBox.Show("No Record Found");
//            con.Close();
//        }

//    }
//}

private void btnProjectDetails_Click(object sender, EventArgs e)
{
    //if (cmbProject.SelectedIndex > 0)projSno ,
    //{
        OleDbConnection con = new OleDbConnection();
        OleDbCommand cmd = new OleDbCommand();
        DataSet ds = new DataSet();
        con.ConnectionString = fnConnectionString();
```

```
            cmd.Connection = con;
            cmd.CommandText = "select BugSno as SNo,(select technique
               from [Technique$] where sno=techSno) as [Technique],
               type as [Category] ,techWeight as [Technique Weight],
               sizeworkProduct as [Size of work Product], FoundDefect
               as [Found Defect] ,perFound as [(%) Found], EscapeDefect
                as [Escaped Defect], perEscape as [(%) Escaped] ,dre as
                [(%) DRE] , removalcost as [Removal Cost], Executontime
                as [Execution time] from [BugEntry$] where isdeleted=
               false and projSno=" + cmbProject.SelectedValue + "";
            try
            {
               OleDbDataAdapter dbAdp = new OleDbDataAdapter(cmd);


               dbAdp.Fill(ds);
               dgBugEntry.DataSource = ds.Tables[0];


               dbAdp.Dispose();




            }
            catch (Exception ex)
            {
               MessageBox.Show("No Record Found");
               con.Close();
            }

        //}
    }

    private void dgBugEntry_CellClick(object sender,
        DataGridViewCellEventArgs e)
    {
        if (e.RowIndex != -1)
        {
            int strValue = 0;
            strValue = e.RowIndex;
            lblHiddenValue.Text = dgBugEntry.Rows[e.RowIndex].Cells[0].
                Value.ToString();
            lblHiddenValue.Visible = false;

            OleDbConnection con = new OleDbConnection();
            OleDbCommand cmd = new OleDbCommand();
            DataSet ds = new DataSet();
            con.ConnectionString = fnConnectionString();
            cmd.Connection = con;
            cmd.CommandText = "select * from [BugEntry$] where
                isDeleted=false and BugSno=" + dgBugEntry.Rows[e.
                RowIndex].Cells[0].Value.ToString();
            try
            {
```

```
OleDbDataAdapter dbAdp = new OleDbDataAdapter(cmd);

dbAdp.Fill(ds);
btnAddBug.Text = "Update";
lblHiddenValue.Text = dgBugEntry.Rows[e.RowIndex].Cells
    [0].Value.ToString();
lblHiddenValue.Visible = false;
btnDelete.Visible = true;

cmbProject.SelectedValue = ds.Tables[0].Rows[0][1].
    ToString();
cmbTechnique.SelectedValue = ds.Tables[0].Rows[0][2].
    ToString();
cmbTechWeight.SelectedItem = ds.Tables[0].Rows[0][3].
    ToString();
txtWeight.Text = ds.Tables[0].Rows[0][4].ToString();
txtSizeofProduct.Text = ds.Tables[0].Rows[0][5].ToString
    ();
txtDefect.Text = ds.Tables[0].Rows[0][6].ToString();
txtPerFoundDefect.Text = ds.Tables[0].Rows[0][7].
    ToString();
txtEscapeDefect.Text = ds.Tables[0].Rows[0][8].ToString
    ();
txtPercEscapeDefect.Text = ds.Tables[0].Rows[0][9].
    ToString();
```

```
                txtDRE.Text = ds.Tables[0].Rows[0][10].ToString();


                txtCost.Text = ds.Tables[0].Rows[0][12].ToString();

                txtTime.Text = ds.Tables[0].Rows[0][13].ToString();

                txtRemovalCost.Text = ds.Tables[0].Rows[0][14].ToString
                    ();

                txtExecutionCost.Text = ds.Tables[0].Rows[0][15].
                    ToString();

                txtPage.Text = ds.Tables[0].Rows[0][11].ToString();




                dbAdp.Dispose();





            }

            catch (Exception ex)

            {


                con.Close();

            }


        }

    }
```

```
        private void txtWeight_TextChanged(object sender, EventArgs e)
        {
            try
            {

            if (txtWeight.Text.Length > 0)
            {
                OleDbConnection con = new OleDbConnection();
                OleDbCommand cmd = new OleDbCommand();
                con.ConnectionString = fnConnectionString();
                con.Open();
                cmd.Connection = con;
                float intValue = 0;
                if (cmbTechWeight.SelectedItem.ToString() != "Select")
                {
                cmd.CommandText = "select " + cmbTechWeight.SelectedItem +
                    "1 from [ProjectDetail$] where sno =" + cmbProject.
                    SelectedValue.ToString() + "";
                intValue = float.Parse(cmd.ExecuteScalar().ToString());
                    intValue= intValue*int.Parse(txtWeight.Text.Trim());
                    intValue = float.Parse(Convert.ToString(intValue / 100 )
                        );
                txtSizeofProduct.Text = Convert.ToString(Decimal.Round(
                    decimal.Parse(intValue.ToString()), 2));
                }
                else
                {
                    txtSizeofProduct.Text = "0";
                }
                 }}
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
                 txtSizeofProduct.Text = "0";
            }



        }

        private void txtDefect_TextChanged(object sender, EventArgs e)
        {
            if (txtDefect.Text.Length > 0)
            {
                if ((txtSizeofProduct.Text.ToString() != "0") && (
                    txtSizeofProduct.Text.Length != 0))
                {
                    float intValue = 0;
                    intValue = float.Parse(txtSizeofProduct.Text.ToString())
                        ;
                    float fValue = 0;
                    fValue =float.Parse(txtDefect.Text.Trim()) /intValue ;
                    fValue =fValue *100;
```

```
txtPerFoundDefect.Text = Convert.ToString( Decimal.Round
    (decimal.Parse(fValue.ToString()), 2) );




if ((txtDefect.Text.ToString() != "0") && (
    txtEscapeDefect.Text.ToString() != "0"))
{
    float fDefect = 0;
    fDefect = float.Parse(txtDefect.Text.Trim());

    float fEscape = 0;
    if (txtEscapeDefect.Text.Length == 0)
    {
        fEscape = 0;
    }
    else
    {
        fEscape = float.Parse(txtEscapeDefect.Text.Trim()
            );
    }



    float fTotal = 0;
    fTotal = fDefect + fEscape;
```

```
                        float fFinal = 0;
                        fFinal = fDefect / fTotal;
                        fFinal = fFinal * 100;

                        txtDRE.Text = Convert.ToString(Decimal.Round(decimal
                            .Parse(fFinal.ToString()), 2));

                    }
                    else
                    {
                        txtDRE.Text = "0";
                    }
                }
                else
                {
                    txtPerFoundDefect.Text = "0";
                    txtDRE.Text = "0";
                }



            }
        }

        private void txtEscapeDefect_TextChanged(object sender, EventArgs e
            )
        {
```

```
if (txtEscapeDefect.Text.Length > 0)
{
    if ((txtSizeofProduct.Text.ToString() != "0") && (
        txtSizeofProduct.Text.Length != 0))
    {
        float intValue = 0;
        intValue = float.Parse(txtSizeofProduct.Text.ToString())
            ;
        float fValue = 0;
        fValue = float.Parse(txtEscapeDefect.Text.Trim()) /
            intValue;
        fValue = fValue * 100;
        txtPercEscapeDefect.Text = Convert.ToString(Decimal.
            Round(decimal.Parse(fValue.ToString()), 2));



        if ((txtDefect.Text.ToString() != "0") && (
            txtEscapeDefect.Text.ToString() != "0"))
        {
            float fDefect = 0;
            fDefect = float.Parse(txtDefect.Text.Trim());

            float fEscape = 0;
            fEscape = float.Parse(txtEscapeDefect.Text.Trim());
            float fTotal = 0;
            fTotal = fDefect + fEscape;

            float fFinal = 0;
            fFinal = fDefect / fTotal;
            fFinal = fFinal * 100;

            txtDRE.Text = Convert.ToString(Decimal.Round(decimal
                .Parse(fFinal.ToString()), 2));

        }
        else
        {
            txtDRE.Text = "0";
        }


    }
    else
    {
        txtPercEscapeDefect.Text = "0";
        txtDRE.Text = "0";
    }


}
```

```
}

private void btnCancel_Click(object sender, EventArgs e)
{
    cmbTechWeight.SelectedIndex = 0;
    txtDefect.Text = "0";
    txtEscapeDefect.Text = "0";
    txtPercEscapeDefect.Text = "0";
    txtPerFoundDefect.Text = "0";
    txtSizeofProduct.Text = "0";
    txtWeight.Text = "0";

    txtDRE.Text = "0";
    txtPage.Text = "0";
    txtCost.Text = "0";
    txtTime.Text = "0";
    txtExecutionCost.Text = "0";
    txtRemovalCost.Text = "0";

    btnAddBug.Text = "Add";
    lblHiddenValue.Text = "";
}

private void txtCost_TextChanged(object sender, EventArgs e)
```

```
{
    if (txtPage.Text.Length > 0)
    {
        if (txtDefect.Text.ToString() != "0")
        {
            int intValue = 0;
            intValue = int.Parse(txtDefect.Text.ToString());
            float fnValue =0;
            fnValue = float.Parse(txtCost.Text.Trim()) / intValue;
            txtRemovalCost.Text = Convert.ToString(fnValue);

        }
        else
        {
            //txtCost.Text = "0";
        }
    }

}

private void txtTime_TextChanged(object sender, EventArgs e)
{
    if (txtPage.Text.Length > 0)
    {
        if (txtPage.Text.ToString() != "0")
        {
            float intValue = 0;
            intValue = float.Parse(txtPage.Text.ToString());
            txtExecutionCost.Text = Convert.ToString(float.Parse(
                txtTime.Text.Trim()) / intValue);
        }
        else
        {
            //txtExecutionCost.Text = "0";
        }
    }
}

private void txtPage_TextChanged(object sender, EventArgs e)
{

    if (txtPage.Text.Length > 0)
    {
        if (txtPage.Text.ToString() != "0")
        {
            int intValue = 0;
            intValue = int.Parse(txtPage.Text.ToString());
            txtExecutionCost.Text = Convert.ToString(int.Parse(
                txtTime.Text.Trim()) / intValue);
            intValue = int.Parse(txtDefect.Text.ToString());
            txtRemovalCost.Text = Convert.ToString(int.Parse(txtCost
                .Text.Trim()) / intValue);
        }
        else
        {
            txtRemovalCost.Text = "0";
            txtExecutionCost.Text = "0";
        }
    }
}

private void btnDelete_Click(object sender, EventArgs e)
{
    OleDbConnection con = new OleDbConnection();
    OleDbCommand cmd = new OleDbCommand();

    try
    {
        if (cmbTechWeight.SelectedIndex > 0)
        {
            if (lblHiddenValue.Text.Length > 0)
            {
                con.ConnectionString = fnConnectionString();
                con.Open();
                cmd.Connection = con;
                int intValue = 0;
                intValue = int.Parse(lblHiddenValue.Text);

                cmd.CommandText = "update [BugEntry$] set isdeleted=
                    true where BugSno= " + intValue;
                cmd.ExecuteNonQuery();
                con.Close();
                btnDelete.Visible = false;
                btnAddBug.Text = "Add";
                lblPerFoundDefect.Text = "";
                MessageBox.Show("Record Deleted Successfully");
            }

            ShowDate();
            txtDefect.Text = "0";
            txtEscapeDefect.Text = "0";
            txtPercEscapeDefect.Text = "0";
            txtPerFoundDefect.Text = "0";
            txtSizeofProduct.Text = "0";
            txtWeight.Text = "0";

            txtDRE.Text = "0";
            txtPage.Text = "0";
            txtCost.Text = "0";
            txtTime.Text = "0";
            txtExecutionCost.Text = "0";
            txtRemovalCost.Text = "0";
            btnProjectDetails_Click(null, null);


        }
        else
        {
            MessageBox.Show("Select Bug Entry");


        }




     }
     catch (Exception ex)
     {
        MessageBox.Show(ex.Message);
        con.Close();
     }
 }

 private void txtWeight_KeyPress(object sender, KeyPressEventArgs e)
 {
     if (!char.IsControl(e.KeyChar) && !char.IsDigit(e.KeyChar) && e
        .KeyChar != '.')
     {
        e.Handled = true;
     }
     // only allow one decimal point
     if (e.KeyChar == '.'
        && (sender as TextBox).Text.IndexOf('.') > -1)
     {
        e.Handled = true;
     }
 }

 private void txtDefect_KeyPress(object sender, KeyPressEventArgs e)
 {
     if (!char.IsControl(e.KeyChar) && !char.IsDigit(e.KeyChar) && e
        .KeyChar != '.')
     {
        e.Handled = true;
     }

     // only allow one decimal point
     if (e.KeyChar == '.'
        && (sender as TextBox).Text.IndexOf('.') > -1)
     {
        e.Handled = true;
     }
 }

 private void txtEscapeDefect_KeyPress(object sender,
     KeyPressEventArgs e)
 {
     if (!char.IsControl(e.KeyChar) && !char.IsDigit(e.KeyChar) && e
        .KeyChar != '.')
     {
        e.Handled = true;
     }

     // only allow one decimal point
     if (e.KeyChar == '.'
        && (sender as TextBox).Text.IndexOf('.') > -1)
     {
        e.Handled = true;
     }
 }

 private void txtCost_KeyPress(object sender, KeyPressEventArgs e)
 {
     if (!char.IsControl(e.KeyChar) && !char.IsDigit(e.KeyChar) && e
        .KeyChar != '.')
     {
        e.Handled = true;
     }

     // only allow one decimal point
     if (e.KeyChar == '.'
 && (sender as TextBox).Text.IndexOf('.') > -1)
{
 e.Handled = true;
}
}

private void txtTime_KeyPress(object sender, KeyPressEventArgs e)
{
if (!char.IsControl(e.KeyChar) && !char.IsDigit(e.KeyChar) && e
 .KeyChar != '.')
{
 e.Handled = true;
}

// only allow one decimal point
if (e.KeyChar == '.'
 && (sender as TextBox).Text.IndexOf('.') > -1)
{
 e.Handled = true;
}
}

private void txtPage_KeyPress(object sender, KeyPressEventArgs e)
{
if (!char.IsControl(e.KeyChar) && !char.IsDigit(e.KeyChar) && e
 .KeyChar != '.')
```

```
            {
                e.Handled = true;
            }

            // only allow one decimal point
            if (e.KeyChar == '.'
                && (sender as TextBox).Text.IndexOf('.') > -1)
            {
                e.Handled = true;
            }
        }

    }
}
```

Listing A.3: Source Code of BugEntry Class

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Data.OleDb;


namespace QA
{
    public partial class Project : Form
    {
       public Project()
       {
           InitializeComponent();
           ShowDate();
       }

       private void Project_Load(object sender, EventArgs e)
       {

       }

       private void btnAdd_Click(object sender, EventArgs e)
       {
           OleDbConnection con = new OleDbConnection();
           OleDbCommand cmd = new OleDbCommand();

           try
           {
               if (txtProjectName.Text.Length > 0)
               {
                   if (label1.Text.Length > 0)
                   {
                       con.ConnectionString = fnConnectionString();
                       con.Open();
                       cmd.Connection = con;
                       int intValue = 0;

                       intValue = int.Parse(label1.Text);
                       float fvalue1 = 0, fvalue2 = 0,fresult=0;

                       fvalue1 = float.Parse(txtArtFact.Text.Trim());
                       fvalue2 = float.Parse(txtHigh.Text.Trim());
                       fresult = (fvalue1 * fvalue2) / 100;
                       lblPHeighValue.Text = Convert.ToString(Decimal.Round
                           (decimal.Parse(fresult.ToString()), 2));

                       fvalue1 = float.Parse(txtArtFact.Text.Trim());
                       fvalue2 = float.Parse(txtMedium.Text.Trim());
                       fresult = (fvalue1 * fvalue2) / 100;
                       lblPMediumValue.Text = Convert.ToString(Decimal.
                           Round(decimal.Parse(fresult.ToString()), 2));

                       fvalue1 = float.Parse(txtArtFact.Text.Trim());
                       fvalue2 = float.Parse(txtLow.Text.Trim());
                       fresult = (fvalue1 * fvalue2) / 100;
                       lblPLowValue.Text = Convert.ToString(Decimal.Round(
               decimal.Parse(fresult.ToString()), 2));




           cmd.CommandText = "update [ProjectDetail$] set
               Projectname ='" + txtProjectName.Text.Trim() + "
               ',phase ='" + txtPhase.Text.Trim() + "',artfact
               ='" + txtArtFact.Text.Trim() + "',high ='" +
               txtHigh.Text.Trim() + "',medium ='" + txtMedium.
               Text.Trim() + "',low ='" + txtLow.Text.Trim() + "
               ' ,high1 =" + lblPHeighValue.Text.Trim() + ",
               medium1 =" + lblPMediumValue.Text.Trim() + ",low1
                =" + lblPLowValue.Text.Trim() + " where sno= " +
                intValue;
           cmd.ExecuteNonQuery();
           con.Close();
           btnDelete.Visible = false;
           btnAdd.Text = "Add";
           label1.Text = "";
       }
       else
       {

           con.ConnectionString = fnConnectionString();
           con.Open();
           cmd.Connection = con;
```

```
int intValue = 0;
cmd.CommandText = "select count(sno) from [
   ProjectDetail$]";
intValue = int.Parse(cmd.ExecuteScalar().ToString())
   ;
if (intValue == 0)
{


}
else
{
    cmd.CommandText = "select max(sno) from [
       ProjectDetail$]";
    intValue = int.Parse(cmd.ExecuteScalar().ToString
       ());
}
intValue = intValue + 1;
float fvalue1 = 0, fvalue2 = 0, fresult = 0;

fvalue1 = float.Parse(txtArtFact.Text.Trim());
fvalue2 = float.Parse(txtHigh.Text.Trim());
fresult = (fvalue1 * fvalue2) / 100;
lblPHeighValue.Text = Convert.ToString(Decimal.Round
   (decimal.Parse(fresult.ToString()), 2));

fvalue1 = float.Parse(txtArtFact.Text.Trim());
```

```
            fvalue2 = float.Parse(txtMedium.Text.Trim());
            fresult = (fvalue1 * fvalue2) / 100;
            lblPMediumValue.Text = Convert.ToString(Decimal.
                Round(decimal.Parse(fresult.ToString()), 2));

            fvalue1 = float.Parse(txtArtFact.Text.Trim());
            fvalue2 = float.Parse(txtLow.Text.Trim());
            fresult = (fvalue1 * fvalue2) / 100;
            lblPLowValue.Text = Convert.ToString(Decimal.Round(
                decimal.Parse(fresult.ToString()), 2));

            cmd.CommandText = "insert into [ProjectDetail$](sno,
                Projectname,phase,artfact,high,medium,low,high1,
                medium1,low1,isDeleted) values ('" + intValue + "
                ','" + txtProjectName.Text.Trim() + "','" +
                txtPhase.Text.Trim() + "','" + txtArtFact.Text.
                Trim() + "','" + txtHigh.Text.Trim() + "','" +
                txtMedium.Text.Trim() + "','" + txtLow.Text.Trim
                () + "','" + lblPHeighValue .Text.Trim() + "','"
                + lblPMediumValue.Text.Trim() + "','" +
                lblPLowValue.Text.Trim() + "',false)";
            cmd.ExecuteNonQuery();
            btnDelete.Visible = false;
            con.Close();
        }
        ShowDate();
```

```
                txtProjectName.Text = "";

                txtPhase.Text = "";

                txtArtFact.Text = "";

                txtHigh.Text = "";

                txtMedium.Text = "";

                txtLow.Text = "";

                lblPHeighValue.Text = "0";

                lblPMediumValue.Text = "0";

                lblPLowValue.Text = "0";


            }

            else

            {

                MessageBox.Show("Enter Technique");


            }




        }

        catch (Exception ex)

        {

            MessageBox.Show(ex.Message);

            con.Close();

        }

    }
```

```
private string fnConnectionString()

{

    string strConnection;

    strConnection = Properties.Settings.Default.cString;

    string strPath;

    strPath = System.IO.Directory.GetCurrentDirectory();

    strPath = strPath + "\\" + strConnection;

    strConnection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source='

        " + strPath + "';Extended Properties='Excel 8.0;HDR=YES;'";


    return strConnection;

}

private void ShowDate()

{


    OleDbConnection con = new OleDbConnection();

    OleDbCommand cmd = new OleDbCommand();

    DataSet ds = new DataSet();

    con.ConnectionString = fnConnectionString();

    cmd.Connection = con;

    cmd.CommandText = "select sno,Projectname,phase,artfact,high,

        medium,low from [ProjectDetail$] where isDeleted=false";

    try

    {

        OleDbDataAdapter dbAdp = new OleDbDataAdapter(cmd);

        //dbAdp.SelectCommand = "select * from [Technique$]";
```

```
            dbAdp.Fill(ds);

            dgProjectDetails.DataSource = ds.Tables[0];


            dbAdp.Dispose();




        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
            con.Close();
        }



    }
    private void btnRefresh_Click(object sender, EventArgs e)
    {
        ShowDate();
    }

    private void dgProjectDetails_CellClick(object sender,
        DataGridViewCellEventArgs e)
    {

            int strValue = 0;
```

```
              strValue = e.RowIndex;


              txtProjectName.Text = dgProjectDetails.Rows[e.RowIndex].
                 Cells[1].Value.ToString();
              txtPhase.Text = dgProjectDetails.Rows[e.RowIndex].Cells[2].
                 Value.ToString();
              txtArtFact.Text = dgProjectDetails.Rows[e.RowIndex].Cells
                 [3].Value.ToString();
              txtHigh.Text = dgProjectDetails.Rows[e.RowIndex].Cells[4].
                 Value.ToString();
              txtMedium.Text = dgProjectDetails.Rows[e.RowIndex].Cells
                 [5].Value.ToString();
              txtLow.Text = dgProjectDetails.Rows[e.RowIndex].Cells[6].
                 Value.ToString();


              //lblPHeighValue.Text = dgProjectDetails.Rows[e.RowIndex].
                 Cells[7].Value.ToString();
              //lblPMediumValue.Text = dgProjectDetails.Rows[e.RowIndex].
                 Cells[8].Value.ToString();
              //lblPLowValue.Text = dgProjectDetails.Rows[e.RowIndex].
                 Cells[9].Value.ToString();


              btnAdd .Text = "Update";
              label1.Text = dgProjectDetails.Rows[e.RowIndex].Cells[0].
                 Value.ToString();
              label1.Visible = false;
```

```
        btnDelete.Visible = true;





    }


    private void btnDelete_Click(object sender, EventArgs e)
    {
        OleDbConnection con = new OleDbConnection();
        OleDbCommand cmd = new OleDbCommand();

        try
        {
            if (txtProjectName.Text.Length > 0)
            {
                if (label1.Text.Length > 0)
                {
                    con.ConnectionString = fnConnectionString();
                    con.Open();
                    cmd.Connection = con;
                    int intValue = 0;

                    intValue = int.Parse(label1.Text);

                    cmd.CommandText = "update [ProjectDetail$] set
                    isdeleted=true where sno= " + intValue;
                cmd.ExecuteNonQuery();

                cmd.CommandText = "update [BugEntry$] set isdeleted=
                    true where projSno= " + intValue;
                cmd.ExecuteNonQuery();
                con.Close();
                btnDelete.Visible = false;
                btnAdd.Text = "Add";
                label1.Text = "";
                MessageBox.Show("Record Deleted Successfully");
            }

            ShowDate();
            txtProjectName.Text = "";
            txtPhase.Text = "";
            txtArtFact.Text = "";
            txtHigh.Text = "";
            txtMedium.Text = "";
            txtLow.Text = "";
            lblPHeighValue.Text = "0";
            lblPMediumValue.Text = "0";
            lblPLowValue.Text = "0";

        }
        else
```

```
            {
                MessageBox.Show("Enter Technique");

            }




        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
            con.Close();
        }
    }

    private void txtHigh_KeyPress(object sender, KeyPressEventArgs e)
    {
        if (!char.IsControl(e.KeyChar) && !char.IsDigit(e.KeyChar) && e
            .KeyChar != '.')
        {
            e.Handled = true;
        }

        // only allow one decimal point
        if (e.KeyChar == '.'
            && (sender as TextBox).Text.IndexOf('.') > -1)
        {
            e.Handled = true;
        }


    }

    private void txtMedium_KeyPress(object sender, KeyPressEventArgs e)
    {
        if (!char.IsControl(e.KeyChar) && !char.IsDigit(e.KeyChar) && e
            .KeyChar != '.')
        {
            e.Handled = true;
        }

        // only allow one decimal point
        if (e.KeyChar == '.'
            && (sender as TextBox).Text.IndexOf('.') > -1)
        {
            e.Handled = true;
        }
    }

    private void txtLow_KeyPress(object sender, KeyPressEventArgs e)
    {
        if (!char.IsControl(e.KeyChar) && !char.IsDigit(e.KeyChar) && e
            .KeyChar != '.')
        {
            e.Handled = true;
        }

        // only allow one decimal point
        if (e.KeyChar == '.'
            && (sender as TextBox).Text.IndexOf('.') > -1)
        {
            e.Handled = true;
        }
    }

    private void txtArtFact_KeyPress(object sender, KeyPressEventArgs e
        )
    {
        if (!char.IsControl(e.KeyChar) && !char.IsDigit(e.KeyChar) && e
            .KeyChar != '.')
        {
            e.Handled = true;
        }

        // only allow one decimal point
        if (e.KeyChar == '.'
            && (sender as TextBox).Text.IndexOf('.') > -1)
        {
            e.Handled = true;
        }
    }


    private void btnScenerioHeigh_Click(object sender, EventArgs e)
    {
        if ((txtLabourRate.Text.Length > 0) && (txtHigh.Text.Length >
            0) && (txtDefectEscalationCost.Text.Length > 0) && (
            txtArtFact.Text.Length > 0))
        {
            float fValue1, fValue2,fResult;
            fResult = 0;
            fValue1 = float.Parse(txtArtFact.Text.Trim());
            fValue2 = float.Parse(txtHigh.Text.Trim());
            fResult = ((fValue1 * fValue2) / 100);
            string sResult = "0";
            sResult = Convert.ToString(Decimal.Round(decimal.Parse(
                fResult.ToString()), 2));
            Scenerio mn = new Scenerio(txtHigh.Text, txtLabourRate.Text
                , txtDefectEscalationCost.Text, sResult, fValue1.
                ToString());
            mn.Show();

        }
        else
        {
            MessageBox.Show("Please Enter Require Data");
        }
    }

    private void txtLabourRate_KeyPress(object sender,
        KeyPressEventArgs e)
    {
        if (!char.IsControl(e.KeyChar) && !char.IsDigit(e.KeyChar) && e
            .KeyChar != '.')
        {
            e.Handled = true;
        }

        // only allow one decimal point
        if (e.KeyChar == '.'
            && (sender as TextBox).Text.IndexOf('.') > -1)
        {
            e.Handled = true;
        }
    }

    private void txtDefectEscalationCost_KeyPress(object sender,
        KeyPressEventArgs e)
    {
        if (!char.IsControl(e.KeyChar) && !char.IsDigit(e.KeyChar) && e
            .KeyChar != '.')
        {
                e.Handled = true;
            }


            // only allow one decimal point
            if (e.KeyChar == '.'
                && (sender as TextBox).Text.IndexOf('.') > -1)
            {
                e.Handled = true;
            }
        }






    }
}
```

Listing A.4: Source Code of Project Class

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
//using Excel = Microsoft.Office.Interop.Excel;
using System.Reflection;

using System.IO;
using System.Runtime.InteropServices;




namespace QA
{
    public partial class Login : Form
    {
        public Login()
        {
            InitializeComponent();
        }

        private void Login_Load(object sender, EventArgs e)
        {

        }

        private void btnAddTechnique_Click(object sender, EventArgs e)
        {
            Main mn= new Main ();
            mn.Show();
        }

        private void btnProject_Click(object sender, EventArgs e)
        {
            Project mn = new Project();
            mn.Show();
        }

        private void btnBugEntry_Click(object sender, EventArgs e)
        {
            BugEntry mn = new BugEntry();
            mn.Show();
        }

        private void btnTechnique_Click(object sender, EventArgs e)
        {
            PhaseReport mn = new PhaseReport();
            mn.Show();
        }

        private void btnCostTime_Click(object sender, EventArgs e)
        {
            CostTime mn = new CostTime();
            mn.Show();
        }


        private void btnSizeWorkProduct_Click(object sender, EventArgs e)
        {
            WorkProduct mn = new WorkProduct();
            mn.Show();
        }
        private static string GetMacro()
        {
            StringBuilder sb = new StringBuilder();

            sb.Append("Sub FormatSheet()" + "\n");
            sb.Append(" Range(\"A1:D1\").Select " + "\n");
            sb.Append(" Selection.Font.ColorIndex = 3" + "\n");
            sb.Append("End Sub");

            return sb.ToString();
        }
        }
}
```

Listing A.5: Source Code of CostTime Class

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Data.OleDb;

namespace QA
{
    public partial class WorkProduct : Form
    {
        public WorkProduct()
        {
            InitializeComponent();
        }
    private void WorkProduct_Load(object sender, EventArgs e)
    {
        ShowDate();
    }
    private string fnConnectionString()
    {
        string strConnection;
        strConnection = Properties.Settings.Default.cString;
        string strPath;
        strPath = System.IO.Directory.GetCurrentDirectory();
        strPath = strPath + "\\" + strConnection;
        strConnection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source='
            " + strPath + "';Extended Properties='Excel 8.0;HDR=YES;'";

        return strConnection;
    }
    private void ShowDate()
    {

        //OleDbConnection con = new OleDbConnection();
        //OleDbCommand cmd = new OleDbCommand();
        //DataSet ds = new DataSet();
        //DataSet ds2 = new DataSet();
        //con.ConnectionString = fnConnectionString();
        //cmd.Connection = con;
        //cmd.CommandText = "select sno,technique from [Technique$]
```

```
            where isDeleted=false ";
        try
        {
            //OleDbDataAdapter dbAdp = new OleDbDataAdapter(cmd);
            //dbAdp.Fill(ds);

            //cmbTechnique.DataSource = ds.Tables[0];
            //cmbTechnique.DisplayMember = "technique";
            //cmbTechnique.ValueMember = "sno";
            //// cmbTechnique.Items.Insert(0, "Select");
            ////cmbTechnique.Items.Insert(0,string.Empty );
            ////cmbTechWeight.SelectedIndex =-1;
            //dbAdp.Dispose();

            //cmd.CommandText = "select sno,Projectname from [
                ProjectDetail$]";

            //OleDbDataAdapter dbAdpSecond = new OleDbDataAdapter(cmd);
            //dbAdpSecond.Fill(ds2);

            //cmbProject.DataSource = ds2.Tables[0];
            //cmbProject.DisplayMember = "Projectname";
            //cmbProject.ValueMember = "sno";

            ////cmbProject.Items.Insert(0, "Select");
            ////cmbTechWeight.SelectedIndex = 0;
            //dbAdpSecond.Dispose();


            cmbType.Items.Add("High");

            cmbType.Items.Add("Medium");

            cmbType.Items.Add("Low");

            cmbType.Items.Insert(0, "Select");

            cmbType.SelectedIndex = 0;








        }

        catch (Exception ex)

        {

            MessageBox.Show(ex.Message);

            //con.Close();

        }



    }


    private void btnShow_Click(object sender, EventArgs e)
    {
        OleDbConnection con = new OleDbConnection();
        OleDbCommand cmd = new OleDbCommand();
        DataSet ds = new DataSet();
       try
        {
           con.ConnectionString = fnConnectionString();
            cmd.Connection = con;
        if (cmbType.SelectedItem.ToString() != "Select")
        {

            //cmd.CommandText = "select (select Projectname from [
                ProjectDetail$] where sno=projSno) as [Project Name],
                sum(sizeworkProduct) as [Size of Work Product] , (sum(
                FoundDefect) + sum( EscapeDefect)) as [Total Defect]
                from [BugEntry$] where isdeleted=false and techSno=" +
                cmbTechnique.SelectedValue.ToString() + " and type='" +
                cmbType.SelectedItem + "' group by projSno";
            cmd.CommandText = "select (select Projectname from [
                ProjectDetail$] where sno=projSno) as [Project Name],
                sum(sizeworkProduct) as [Size of Work Product] , (sum(
                FoundDefect) + sum( EscapeDefect)) as [Total Defect]
                from [BugEntry$] where isdeleted=false and type='" +
                cmbType.SelectedItem + "' group by projSno";
            OleDbDataAdapter dbAdp = new OleDbDataAdapter(cmd);
            dbAdp.Fill(ds);
```

```
                int i = 0;

                double fresult = 0;



                if (txtConstantValue.Text.Length > 0)

                {

                    fresult = fnRegression(ds, double.Parse(txtConstantValue

                        .Text));


                }

                txtTotalDefect.Text = Convert.ToString(Decimal.Round(

                    decimal.Parse(fresult.ToString()), 2));


                dgWorkSizeproduct.DataSource = ds.Tables[0];

                 dbAdp.Dispose();


             }

             else

             {

                 MessageBox.Show("Please Select the Details");

             }






            }
```

```
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
            con.Close();
        }
    }

    private void txtConstantValue_KeyPress(object sender,
        KeyPressEventArgs e)
    {
        if (!char.IsControl(e.KeyChar) && !char.IsDigit(e.KeyChar) && e
            .KeyChar != '.')
        {
            e.Handled = true;
        }

        // only allow one decimal point
        if (e.KeyChar == '.'
            && (sender as TextBox).Text.IndexOf('.') > -1)
        {
            e.Handled = true;
        }
    }

    private void btnShowDetails_Click(object sender, EventArgs e)
    {
```

```
OleDbConnection con = new OleDbConnection();
OleDbCommand cmd = new OleDbCommand();
DataSet ds = new DataSet();
try
{
    con.ConnectionString = fnConnectionString();
    cmd.Connection = con;
    if (cmbType.SelectedItem.ToString() != "Select")
    {

        //cmd.CommandText = "select (select Projectname from [
            ProjectDetail$] where sno=projSno) as [Project Name
            ], sum(sizeworkProduct) as [Size of Work Product] ,
            (sum(FoundDefect) + sum( EscapeDefect)) as [Total
            Defect] from [BugEntry$] where isdeleted=false and
            techSno=" + cmbTechnique.SelectedValue.ToString() +
            " and type='" + cmbType.SelectedItem + "' group by
            projSno";
        cmd.CommandText = "select (select Projectname from [
            ProjectDetail$] where sno=projSno) as [Project Name
            ], sum(sizeworkProduct) as [Size of Work Product] ,
            (sum(FoundDefect) + sum( EscapeDefect)) as [Total
            Defect] from [BugEntry$] where isdeleted=false and
            type='" + cmbType.SelectedItem + "' group by projSno
            ";
        OleDbDataAdapter dbAdp = new OleDbDataAdapter(cmd);
```

```
                dbAdp.Fill(ds);



                txtTotalDefect.Text = "0";
                dgWorkSizeproduct.DataSource = ds.Tables[0];
                dbAdp.Dispose();

            }
            else
            {
                MessageBox.Show("Please Select the Details");
            }




        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
            con.Close();
        }
    }

    public double fnRegression(DataSet ds,double x)
    {
```

```
double y = 0;

double m = 0;

double c = 0;



int i = 0;


double SumX = 0;


double SumY = 0;


double SumX2 = 0;


double SumXY = 0;


double D = 0;



if (ds.Tables[0].Rows.Count>0)

{


for (i = 0; i < ds.Tables[0].Rows.Count; i++)

{


    SumX += double.Parse(ds.Tables[0].Rows[i][1].ToString().

        Trim());
```

```
            SumY += double.Parse(ds.Tables[0].Rows[i][2].ToString().
                Trim());



            SumX2 += double.Parse(ds.Tables[0].Rows[i][1].ToString().
                Trim()) * double.Parse(ds.Tables[0].Rows[i][1].ToString
                ().Trim());

            SumXY += double.Parse(ds.Tables[0].Rows[i][1].ToString().
                Trim()) * double.Parse(ds.Tables[0].Rows[i][2].ToString
                ().Trim());

        }

        D = ds.Tables[0].Rows.Count * SumX2 - SumX * SumX;

        c = (SumY * SumX2 - SumXY * SumX) / D; //Intercept

        m = (ds.Tables[0].Rows.Count * SumXY - SumY * SumX) / D; //
            Slope

        y = (m * x) + c;
    }
        return y;
    }
    }
}
```

Listing A.6: Source Code of WorkProduct Class