

Best Software Test & Quality Assurance Practices in the project Life-cycle

An approach to the creation of a process for improved test & quality assurance
practices in the project life-cycle of an SME.

Mark Kevitt, BSc

University: Dublin City University
Supervisor: Renaat Verbruggen
School Computer Applications

April 2008

Abstract

The cost of software problems or errors is a significant problem to global industry, not only to the producers of the software but also to their customers and end users of the software.

There is a cost associated with the lack of quality of software to companies who purchase a software product and also to the companies who produce the same piece of software. The task of improving quality on a limited cost base is a difficult one.

The foundation of this thesis lies with the difficult task of evaluating software from its inception through its development until its testing and subsequent release. The focus of this thesis is on the improvement of the testing & quality assurance task in an Irish SME company with software quality problems but with a limited budget.

Testing practices and quality assurance methods are outlined in the thesis explaining what was used during the software quality improvement process in the company. Projects conducted in the company are used for the research in the thesis. Following the quality improvement process in the company a framework for improving software quality was produced and subsequently used and evaluated in another company.

Table of contents

1	Chapter One – Introduction	5
1.1	A software company with software quality problems	5
	Aim	5
	Objectives	6
2	Chapter Two - Methodology.....	7
2.1	Action Research	7
2.2	The type of Action research used in this thesis.....	9
3	Chapter Three - What is software testing	11
3.1.1	Principles of software testing	11
3.2	Principal testing methods	12
3.2.1	Functional testing (black box)	12
3.2.2	Structural testing (white box).....	14
3.2.3	Grey box testing	15
3.2.4	Thread Testing	16
3.2.5	System Testing	17
3.3	The Test Process	20
3.3.1	Test Planning	21
3.3.2	Manage Test Execution.....	24
3.4	Summary	35
4	Chapter Four – Quality Assurance.....	37
4.1	Complications with software and its quality assurance	37
4.1.1	Factors that impact Software Quality	39
4.2	Software Quality Assurance	43
4.2.1	Verification versus Validation	45
4.3	Software Quality measurement.....	46
4.3.1	Software Defects	48
4.3.2	Classification of Software Errors.....	49
4.4	Structure of Software Quality Assurance (SQA).....	52
4.4.1	Planning from the project initiation and project planning stage	53
4.4.2	Management of the Project life-cycle activities and components	57
4.4.3	Defect Prevention process.....	65
4.4.4	Capturing and analysing defect metrics	67
4.4.5	Quality Management Models.....	72
4.4.6	In process metrics for software testing	74
4.5	Refactoring the Management of all SQA components	77
4.5.1	Software Quality Management	77
4.5.2	The SEI Process Capability Maturity model	79
4.5.3	Software Process Assessment	81
4.5.4	Software Quality Auditing	83
4.6	Summary	84
5	Chapter Five – Software Test and Quality Assurance Practice Improvement.....	85
5.1	The first steps to test and QA practice improvements	85
5.1.1	Industry background	86
5.1.2	Description of company X BMS system	87
5.1.3	Research and Development department description.....	90
5.2	The Quality problem that is to be tackled.....	93
5.2.1	The investigation.....	94
5.2.2	The investigation findings.....	95

5.2.3	The proposal to the company	105
5.3	My proposed solution	106
5.3.1	The Principal design factors behind my proposed solution	106
5.3.2	An Initial model	109
5.4	Summary	114
6	Chapter Six - Implementation of improvements.....	115
6.1	Company X - HVAC Controller Project.....	117
6.1.1	HVAC Project description	117
6.1.2	HVAC Plan	118
6.1.3	HVAC Project Implementation (Execution and Observation)	127
6.1.4	HVAC controller project reflection	134
6.2	Company X CNET Project Repeat improvements	136
6.2.1	CNET Project description	136
6.2.2	CNET Plan	137
6.2.3	CNET Project Implementation (Execution and Observation)	139
6.2.4	CNET Controller Project Reflection.....	142
6.3	Company X – UEC8 Engineering Application.....	144
6.3.1	UEC8 Project description	144
6.3.2	UEC8 Plan	145
6.3.3	UEC8 Project Implementation (Execution and Observation).....	146
6.3.4	UEC8 project reflection	149
7	Chapter Seven - Development of a Framework.....	151
7.1	Evolved quality assurance framework	154
7.2	Secondary Independent Assessment of my proposed Solution	160
7.3	Company Y - Project FIIS Application	162
7.3.1	FIIS Project description	162
7.3.2	FIIS Plan	163
7.3.3	FIIS Project Implementation (Execution and Observation).....	166
7.3.4	FIIS project reflection	169
7.4	Summary	170
8	Chapter Eight - Conclusions and Further work	171
8.1	Conclusion	171
8.1.1	Limitations of research	173
8.1.2	Improvements to practices	174
8.2	Further Work.....	175
9	Appendices, Glossary and Bibliographies	176
9.1	Appendix A Company X process documentation.....	176
9.2	Appendix B – Glossary of terms.....	177
9.3	Bibliography	179
9.4	Web Bibliography	180

1 Chapter One – Introduction

1.1 A software company with software quality problems

This thesis is focused on the creation and provision of a testing & quality assurance (QA) process for software quality improvement in an Irish company (the company) and also for the creation of a framework for similar quality improvements in the process for other company's.

Employed in the company as a testing professional I have the responsibility to lead a test department and to ensure that the software released to the customers is of the highest standard. To raise the bar on this standard I decided to conduct research into testing and QA practices and to implement improved practices within the company.

This thesis is a product of the research into test and QA practices and for the provision of an improved test process in the company. This process will combine elements of testing and QA into one process, this one process in turn will be inserted into the company's development lifecycle.

The research was agreed with academic representatives from DCU University and with senior management from the company. I conducted this research on a part time basis with the University while working full time in the company.

Aim

The aim of this thesis is to investigate the best test and QA practices in industry and to design and evaluate a process for implementing best practices in the software lifecycle of a small to medium enterprise (SME) over successive projects.

Objectives

There are a number of objectives for this paper, the first is to define the principles of software testing, describe the numerous testing methodologies and how to effectively conduct this testing on projects in industry. This is covered in the third chapter.

The second objective is to evaluate what constitutes software quality and what factors affect this quality and how, when and where QA can be used in the project life-cycle for improving product quality. This is covered in the fourth chapter.

The third objective is to outline the test and QA effort during a project in a particular company and to evaluate the adoption of improved practices during subsequent projects in the same company. These two topics are covered in the fifth and sixth chapters respectively.

The fourth objective is to develop the improved practices into a framework for evaluation in other company's. This is covered in the seventh chapter.

2 Chapter Two - Methodology

2.1 Action Research

The research methodology that was chosen for this project is action research. Action research is a methodology which has the dual aims of action and research. The action is to bring about change in some community or organisation, and the form of research intended to have both action and research outcomes. The purpose of action research is to learn from your experience, and apply that learning to bringing about change. “The task of the practitioner researcher is to provide leadership and direction to other participants or stakeholders in the research process” (Ernest Stringer. 1996)

Action research in the organisation (David Coughlan et al. 2005)

1. Review current practice
2. Identify an aspect that needs improvement
3. Plan an action
4. Act it out
5. Evaluate the result
6. Re-plan an additional cycle
7. Continue until complete

Examples of Action Research

Action Research, as described by Lewin, proceeds in a spiral of steps composed of planning, action and an evaluation of the result of the action.

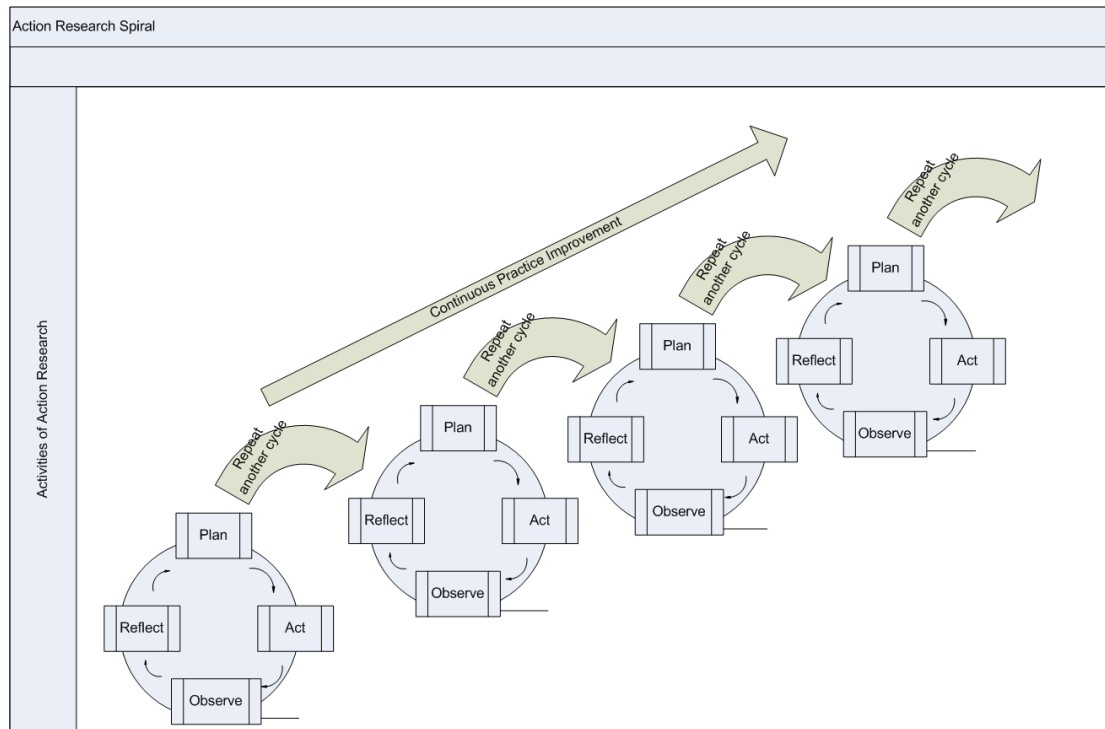


Figure 2.1. The Action Research spiral model.

The advantages of action research are that it lends itself to use in work or community situations. Practitioners, people who work as agents of change, can use it as part of their normal activities. This means that in the course of researching best practices in software quality improvements, it can also be applied during the operation of an organisation.

The disadvantages to action research are that it is harder to do than conventional research. There is a dual role of the researcher to conduct research but also to make changes and record the results of these changes.

2.2 The type of Action research used in this thesis

Holter and Schwartz-Barcott (1993:301) discuss three types of action research, that of a technical collaborative approach, a mutual collaborative approach and an enhancement approach. McKernan (1991:16 -27) lists three types of action research, the three fall roughly into the same categories.

The type of action research that has been chosen for this thesis is that of type I, Technical/Technical-Collaborative. The reason behind this choice is that it closely matches the aims for the thesis. The research includes process improvement and the derivation of a framework for best test and QA practices and to evaluate this framework in a real software project life-cycle.

Type 1: Technical/Technical-Collaborative

(McKernan 1991:16) The underlying goal of the researcher in this approach is to test a particular intervention based on a pre-specified theoretical framework, the nature of the collaboration between the researcher and the practitioner is technical and facilitatory.

Technical action research promotes more efficient and effective practice. It is product directed but promotes personal participation by practitioners in the process of improvement.

Data Collection Methods

Two different data collection methods will be implemented to conduct the research. Both quantitative and qualitative styles are applied to corroborate the data collected. A quantitative experimentation will be the primary method. This will provide statistical data for evaluating the effectiveness of the test & QA practices.

The data will be in the format of the time to complete the project and the cause of any delays if any, the number of test cycles that had to be run and the number of defects found during the testing of the project and their severity. In order to reduce the influences of external dependent variables a secondary technique of interviewing will be conducted.

Population

For this thesis the population will be the employed developers, testers, technical support engineers and managers of the projects in which the experiments are being conducted.

3 Chapter Three - What is software testing

3.1.1 Principles of software testing

The purpose of software testing is to detect errors in the software. The tester should ideally detect all errors before the software is released to the customer. Full test coverage of a program is impossible. “Proving that a program is fault free is equivalent to the famous halting problem of computer science, which is known to be impossible” (Paul C. Jorgensen. 1995)

The main principle of software testing “is the process of executing a program with the intent of finding errors”. (Glenford J. Myers, 2004). To test the program more thoroughly a tester would need to evaluate the program to detect both types of errors. This principle is thus more detailed to “Test the program to see if it does what it is supposed to and to see if it does what it is not supposed to do”. (Glenford J. Myers, 2004)

In order for the tester to find these errors, he will devise a number of tests to execute on the software itself. These tests must be based on prior knowledge of the software. The two main thrusts of testing are firstly based on the composition of the software, i.e. its internal structure. Secondly based on the business or intended purpose of the software, i.e. the functional aspect of the software.

Based on one of these box test paradigms the tester will write a series of tests (test cases) to detect any errors and to evaluate if the outcome of the test meets with the software design. “Invalid and unexpected input data are more effective at detecting errors than valid and expected data” (Glenford J. Myers, 2004). The problem here is determining whether or not the results of the tests are errors or actual expected results.

Where errors are detected, it is prudent to test this area of the program in more detail as statistically more errors will be present in this area “The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section” (Glenford J. Myers, 2004).

3.2 Principal testing methods

3.2.1 Functional testing (black box)

Functional testing is “testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions” (Jerry Zeyu Gao et al, 2003). Functional testing is directed at executing test cases on the functional requirements of software to determine if the results are acceptable.

“The use of equivalence classes as the basis for functional testing has two motivations: we would like to have the sense of complete testing, and at the same time, we would hope that we are avoiding redundancy” (Paul C Jorgensen. 1995)

The method for equivalence classes / partitioning uses two rules:

1. A test case must reduce by more than 1 the number of other test cases that must be developed to achieve some predefined goal of reasonable testing.
2. It covers a large set of other possible test cases, i.e. it tells us something about the presence or absence of errors over and above this specific set of input values.

(Glenford J. Myers, 2004).

The second consideration is used to develop a set of challenging conditions to be tested. The first consideration is then used to develop a minimal set of test cases covering these conditions.

“For the partition testing, input domain will be classified into different disjointed partitions. Ideally, every element in each partition has the same possibility to either reveal or hide a fault. But based on programming experiences, this is usually not true. Values that are close to the boundary of the partition are more likely to expose errors” (Jerry Zeyu Gao et al, 2003).

Boundary value analysis explores test situations on and around the edges of equivalence classes. The conditions are those situations directly on, above and below

the edges of input equivalence classes. The two differences between Equivalent partitioning and boundary analysis are:

1. Boundary value analysis requires that each edge requires a test case, equivalence classes uses one input as one test case.
2. The test cases in boundary analysis require the output space to be considered also for test cases. The output space of equivalence classes are not considered as test cases.

Typically, a few rules of thumb can be applied so that both equivalent partitioning and boundary analysis can both be applied for test cases that are more comprehensive. The edges that are referred to are generally the upper and lower values permitted by an applications input such as the first and last months of the year. The output domain must also be considered so that the expected output is achieved and also to explore each alternative unexpected output.

If an input condition requires a minimum and maximum range such as that above then use the valid months 1 & 12, also use the invalid months of 0 and 13.
If an input condition specifies a range of values permitted such as between -1000.0 and +1000.0 then use -1000.1, -1000.0, 0, 1000.0 and 1000.1.
If the output domain expects to calculate a person's age based on the input date of birth and the current date then attempt to generate additional invalid output domains such as a 0 age, a negative age and an age in excess of the maximum, 200 years old for example.
If the output domain expects more than one output, for example a date of birth, a current age and a retirement age. Then generate an output domain with 0,1,2,3 and 4 valid and invalid output domains.

Fig 3.1 boundary value analysis examples.

A competent tester will however have the traits of wanting to excel at breaking an application in the most unexpected manner and with increased experience will more than likely be able to create additional test cases to accomplish just this. Error guessing is “is likened to a natural intuition or skill to determine how things work and how best to break them” (Glenford J. Myers, 2004) these additional error guessing test cases can unearth the most unexpected outcomes from systems.

3.2.2 Structural testing (white box)

There are two benefits of structural testing; the first is the creation of test cases based on the logic of the application. The second is the detection of how successful tests are by examining how many different paths through a program were executed.

“In path testing, a major difficulty is that there are too many feasible paths in a program. Path-testing techniques use only structural information to derive a finite subset of those paths, and often it is very difficult to derive an effective subset of paths” (Jerry Zeyu GAO et al, 2003)

The use of test cases based on the logic of programs would require a map of the nodes and connecting paths. You would also need equivalent methodologies to determine what test cases to create or to determine by some metrics how successful the test cases were. To test and evaluate the program the tester should select test data so that each path is covered at least once. This does not guarantee that all errors will be detected since there may be a substantially large number of paths in the programs logic. As each decision on a path has a subsequent decision on the same path the magnitude of nodes and different paths increases from 2^2 to 2^n where n is the number of different paths through the code. To increase the rate of error detection a number of metrics can be calculated to evaluate just how successful test cases are.

- Statement coverage
- Decision Coverage
- Condition Coverage
- Decision-condition coverage

The complexity of the logic is determined by the number of different nodes and the number of different possible paths through the application. The use of the above metrics would enable the tester to determine how much of the code has been executed. The test case results demonstrate the likelihood of the future success of the application.

3.2.3 Grey box testing

Grey box testing is a blend of white and black box testing. In order to conduct white box testing the code needs to be analysed and the paths through the logic mapped out. This is a time consuming and expensive process which would typically require tool support. It would be conducted in the more mission critical software systems such as aeronautics, automotive and other mission critical systems. Not all software houses have such tools or the time or need to go to such depths for analysing the code. However ignoring structural testing and only conducting functional tests would leave a large percentage of defects unnoticed until the system goes live. To circumvent this grey box testing is used.

The design or architecture of the system would be used to map out the logic of certain components in the system. The developers themselves would typically also be asked for input into how certain modules were coded. This information is invaluable in assisting the tester design intuitive positive and negative tests for the system. Test data could also be created that would give best coverage of the system.

The data flow and business organisation of the application under test would also greatly assist the tester to ensure that the test cases adequately cover all of the functionality. The design of use cases that depict user scenarios help the tester to appreciate the important business rules and to focus on these. The flow of data during the business functionality is also critical for testing. “Use cases capture the system’s functional requirements from the user’s perspective; they also serve as the foundation for developing system test cases”. (William. E. Lewis. 2005)

3.2.4 Thread Testing

“An approach most suitable for real-time systems is that of thread testing. The system is segmented into threads where software test and construction are interwoven. The modules associated with each thread are coded and tested in the order that is defined within the schedule. Integrating the builds will eventually construct the entire system”. (De Millo et al. 1987).

The feasibility of thread testing is dependent on a sequential development process. In a scheduled sequence the builds of software should deliver a certain component of functionality. The testing is conducted on each successive build or thread, each thread if successful is then integrated into the entire system. The test process is intertwined with the development process more closely than with other approaches. The most critical threads should be developed and tested first. The schedule for both development and test would overlap on the same components with development having a certain amount of lead time. A good visual representation would be a staggered production line, where certain components are assembled in a predefined order with one side of the line assembling the components with the opposite member conducting quality control checks. By the time that the product reaches the end of the line it should be fully complete and approved by quality.

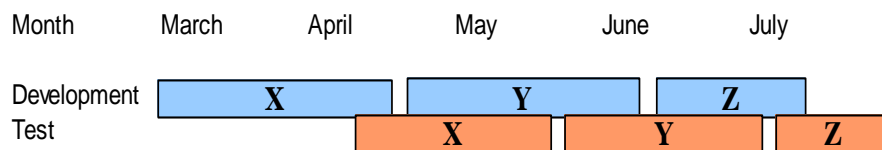


Fig 3.2 Example schedule for thread testing with 3 threads X, Y, and Z

3.2.5 System Testing

Subsequent to integration testing a complete system or application has been developed with working interfaces. This does not mean that the system is necessarily complete. In order to be satisfied that a system is both entirely complete and correct, you would need to be confident that all of its intended functionality exists and that it performs each correctly under every foreseeable circumstance that is possible during its operation. System testing is an attempt to demonstrate if the program as a whole does meet its stated objective.

System testing is non trivial and is therefore broken down into many different test types, sometimes referred to as higher order tests. Each of the higher order tests targets a particular domain of the system. These domains are likely problem areas that could potentially prevent the system performing some of its intended purpose. System testing as its name suggests, means that each of the elected higher order tests are executed on the system as a whole.

It is advantageous for an independent team to perform the system testing including some end users, a representative of the development team and of course the testers who have to know the system in its entirety and the target audience.

“When you finish module-testing a program, you have really only just begun the testing process. This is especially true of large or complex programs. To complete testing, then some form of further testing is necessary. We call this new form *higher-order* testing. The need for higher-order testing increases as the size of the program increases. The reason is that the ratio of design errors to coding errors is considerably higher in large programs than in small programs.” (Glenford J. Myers, 2004).

The main higher order tests are listed and 2 relevant for this thesis are outlined below:

Performance testing
Load/Volume testing
Stress testing
Security testing
Compatibility testing
Conversion Testing
Backup testing
Recovery testing
Installation testing
Reliability testing
Usability testing
Acceptance testing
Functional Testing

Fig 3.3 Higher order system tests

Two system tests that are pertinent to this thesis are explained below in more detail.

Usability Tests

The objective of usability testing is to determine how well the user will be able to use, understand and navigate through the application. If the system has a UI that is separate from the main thrust of the business rules and data then usability testing should be performed on the UI as early as one is available. If the UI is integrated with the entire application then changes to it are very costly and if possible as portions of the UI are developed they should be evaluated for their usability. Without consideration of the type of user interface employed there are a common number of considerations that should be used when designing the user interface tests.

The tests should involve assessing the system to check if it has good human compatible interface (HCI) features such as:

- Intuitive to use
- No overly complex prompts or choices
- No non standard UI elements that are unfamiliar to competent users
- Customisable global defaults and options for advanced users
- No poor error messages that are uninformative
- User is not required to remember too much information during navigation
- No difficult log in procedures
- No unclear defaults
- Positive feedback after input completion
- Effective feedback during lengthy processing
- No loss of data during navigation through items
- No unclear position and direction within the navigation through the system
- General Inconsistency
- Clarity of purpose, intentions
- Uniform style and abbreviations

Acceptance Testing

The objective of acceptance testing is for the user to verify that they are satisfied with the system and that they are content that all of the requirements have been met. It is a requirements based test performed by the customer or a subset of end users tests. Depending on the customer the acceptance tests are designed by the QA department or the customer themselves. It is ultimately executed by the customer. The tests are devised to show that the program meet its contracted requirements.

The acceptance tests may be performed on a pre-production environment or on a production environment or even both. Typically the software development contract will state a time frame in which the customer may conduct their acceptance tests and within this time frame the development house is liable for any defects encountered. Outside of this time frame any defects fixes are charged to the customer.

3.3 The Test Process

The creation of tests and the execution of tests is a process itself, it is also a sub process for the entire development effort, “Planning, design and performance of testing activities are carried out throughout the software development process. These activities are divided in phases, beginning in the design stage and ending when the software is installed at the customer’s site”. (Daniel Galin, 2004).

The test process is broken down into activities by individual testers where the main thrust of the test effort is concentrated. It is advisable to document the effort in an evolving timeframe where the planning and preparations are conducted first with the design and execution of test cases later. The management of the test process is crucial for an effective test effort, for individual applications the test process is used on a per project basis.

“Software testing focuses on test planning, test design, test development, and test execution. Quality control is the process and methods used to monitor work and observe whether requirements are met. It focuses on structured walkthroughs and inspections to remove defects introduced during the software development lifecycle” (William E. Lewis, 2004). The topic of quality control or quality assurance is covered in the next chapter. It is noteworthy that in software development company’s there is frequent confusion over the definition of testing and that of quality assurance. The team who perform testing are frequently titled quality assurance or QA, but are responsible for testing only. If the company do not have a dedicated quality assurance team then the testing team can bear this responsibility in addition to testing. It is because of this reason that I would like to combine test and QA practices into one process. This will be covered in chapter six. Currently the traditional testing only process will be discussed.

In the context of testing only, the test process consists of the following elements:

1. Test planning
 - a. Test preparation – test strategy
 - b. Test planning – test plan
 - c. Test design – test scripts
2. Test execution
3. Defect management
4. Release management

3.3.1 Test Planning

It is imperative that proper test planning be conducted from the project outset rather than beginning testing after the code has been completed. The test project planning must coincide with the project plan and there are different stages of the planning process. It is common to have the following test documents in the order below:

1. The test strategy
2. The test plan
3. The master test plan (Should a number of test plans be required)
4. The test cases

1. The test strategy

The test strategy is a document where the entire test approach is outlined and all key people, activities and participants are listed in relation to their responsibilities. The test strategy is written at the beginning of a project where the project stakeholders have been assigned. The person responsible for this document is typically an experienced tester, a leader or manager depending on the size of the project.

The activities associated with the test strategy include the following:

1. Information gathering
 - Interview project stakeholders (role players)

- Understand the project by reading documents e.g. end users, resources, existing application documentation, budget, expected project duration.
2. Identify the project objectives (purpose, scope, benefits, strategy, constraints)
 - understand the project plans (schedule, assignments – resources, project breakdown – modules)
 - understand the project development methodology (how is it being developed, level of expertise)
 - identify high level business requirements (minimum HW requirements, performance requirements, design constraints, dB, platform)
 - Perform risk analysis (compatibility, data loss, user abilities, high risk components, weighting areas of risk)
 3. Document the strategy that defines how the project will be tested under a number of headings.

Content Heading	Purpose
Introduction	Describe the project for the reader
Scope	What is in scope for the test effort
References	Reference external documents
Test Approach	How the testing will be conducted
Test Types	The testing types that will be used
Traceability	How requirements will be tested
Schedule	Identify at a high level the time frame for testing
Roles and Responsibilities	Who will do what
Test Tools	What tools will be needed
Test Environment	What is required for the testing
Test Standards	What standards need to be met
Entry and Exit Criteria	Define what these are

Fig 3.4 Typical Test strategy headings

The Test Plan

The test plan is a document that takes the test strategy and develops a workable plan for the testing that will be executed for the project. The test plan is a lower level document than strategy, the same principles are applied to its creation as a that of a test strategy. It is a document which will change to meet with the projects execution. There would be more project documentation available and more key participants in the project available for information gathering and for the document's approval. The typical headings and contents of the headings follow on the next page:

Content Heading	Purpose
Introduction	Describe the project for the reader
Risks, dependencies, assumptions and constraints	Identify and mitigate against the risks for the testing of the project
Testing Approach	The test stages and test types for each stage
Entry and exit criteria	The entry and exit criteria for each stage
Testing process	Define what process will be carried out for the testing stages
Deliverables	The deliverables from the test stages
Milestones	The milestones for each stage
Schedule	Breakdown the testing milestones into their respective schedules
Environmental	The testing environment
Test Data	Test data required
Responsibilities, staffing and training	The resources and training required
Configuration Management / Version Control	The management of requirement changes and versions of code and builds
Test Case Design	How the test cases will be designed
Change Control	What must be followed if there is a change in requirements or builds
Test Execution Management	How test execution will be managed
Defect Tracking	How defects are tracked with development and test

Content Heading	Purpose
Management process and activities	Communication and escalation of issues and reports with management

Fig 3.5 Typical test plan headings

Test case design and documentation

When the test plan has been completed the test cases are designed for each of the test stages. The test case design should adhere to a standard that is outlined in the test plan. It is important when test design is concerned that traceability to project requirements and design documents is adhered to so that test coverage can be assured when the tests are executed. The test stages should all have their suite of test cases. The test cases should become more high level with each progressive stage of testing.

3.3.2 Manage Test Execution

The management of the test effort is a collaborative process primarily between the test team and the development team but also with the business representative of the product and with the customer. In order to ensure an effective test effort and that all participants understand the activities it is worthwhile to document and publish a process in addition to the project documents. The activities in a typical process are not limited to, but may include the following:

Preparations

1. Organise the team
2. Establish the test environment (tools, network, servers, client boxes)
3. Refactor the test schedule if required
4. Verify that any new requirements are refactored in the test plan/scripts
5. Refine tests and create a new test set
6. Verify the new build and conduct smoke tests
7. Regression test the fixes
8. Report initial test progress to verify that testing has commenced

Test Stage Execution

9. Log defects into a database stating the test number if available for traceability
10. Manage the test execution progress and monitor the metrics
11. Manage the defects with testers and development managers
12. Report the weekly status of tests and defects to management
13. Document the defects and write a test summary report

Post Test Stage Execution

14. Write a test summary report
15. Publish metric graphs
16. Evaluate if the exit criteria are met, if not then prepare for the next test iteration
17. If the exit criteria have been met then continue to release management

Release Management

18. Conduct a project closedown meeting and write up the release notes
19. Document the defect metrics indicating quality
20. Bundle the release and distribute it

Post Project review

21. Write a project summary report
22. Improve the test environment, test data and test procedures

Test Project Management

It is crucial to ensure that the project is delivered as defect free as time and costs permit. High profile business losses due to poor quality systems have increased the profile for testing. The Test Manager and tester must be more focused on improving the quality of deliverables at each stage of the development process. The complexity of the system must be broken down into logical components that can be tackled with ease and with accuracy. Armed with this knowledge the test manager must:

- Create complete and meaningful test plan and test cases.
- Be armed with enough business acumen to substantiate any arguments between defects and the criticality of them if necessary
- Be in a position to offer assistance and guidance to other testers and role holders
- Suggest valid improvements to the system
- Always test for full coverage
- Create a knowledge base that grows with each new project

Test Estimation

Test Estimation effort should concern itself with the following resources:

- The number of testers required
- Cost of hardware necessary for testing
- Cost of software necessary for testing
- Break down the key test objectives into tasks that can have resources assigned to them
- Determine the entry and exit criteria of each task
- Determine what tasks can run concurrently
- Enter the tasks into a project schedule grouping with appropriate time frames
 - Time frames that allow for concurrent activity should be assigned to different testers to allow for early completion
 - Time frames that are dependent on previous tasks should also be scheduled on a finish to start basis, this ensures that the previous task is completed before the subsequent task begins
- Enter in milestones for the end of key activities, e.g. test plan complete, test cases complete

Defect monitoring and management

No software product can be produced with 100% perfection. The product will mature over its life-cycle and the number of defects diminish as it improves with enhancements and corrections, unless it outlives its effective life-cycle and becomes unmaintainable.

Defect reporting and tracking are essential to the test management process. Defects need to be reported as they are found with sufficient information for them to be worthwhile reporting. The defects also need to be assessed after reporting, so that the team can agree on the severity of the defect and its impact on the project.

“An integral part of the tester’s workbench is the quality control function which can identify defects uncovered through testing as well as problems in the testing process itself. Appropriate recording and analysis of these defects is essential to improving the testing process” (William E. Lewis, 2004).

The defect report should contain the following information during its initial report:

- Unique Defect number
- Date
- Tester name
- Product Name
- Component or Module ID
- Build number of product
- Unique Test case number
- Reference to test data used
- Steps for reproduction if different from test case
- Severity
- Defect Category
- Defect Status
- Responsibility for follow up

After follow up:

- Developer’s comments
- Tester’s Comments
- Resolution category

Defect meetings need to be conducted periodically. The test team and development team need to attend to agree on the impact of the defects and on their scheduled resolution. It may be advantageous for the test team to discuss their opinion on the defects before holding the meeting. For a maintenance project it may be necessary for the customer or some-customer facing team to be present for a customer’s perspective or business impact. Typical defect classification, descriptions and resolution priority are listed below.

Classification	Description	Action
Show stopper:	Product cannot be used without this item being fixed, potential data loss or corruption. Users and business use is severely affected. The reputation of the company would certainly be diminished.	Resolve immediately
Critical	There is a workaround that allows for the system to be used. Business use is affected but only minimally. Would affect the reputation of the company.	Resolve as soon as possible.
Normal	If the defect does not fit into either of the above category. Minimum affect on business use.	Resolve when working on the next build.
Minor	Very trivial defect that does not affect business use. Typically a cosmetic issue or something that happens very infrequently.	Resolve when time permits.

Fig 3.6 Defect classification and resolution priorities.

Ideally there would be two methods of reporting the classification of a defect.

- The first is the perception of the defect by the discoverer: a tester, customer or other user. This is considered the defect classification. The impact on the test effort or the customer.
- The second is the agreed impact on the test team, the customer and for the ability of the development team to resolve the defect. This can be considered as the resolution priority.

When a defect state is changed for example by altering its classification or when it is resolved then an entry should be made for it in the repository. The purpose of changing its state should be documented along with the individual responsible for altering its state. This aids traceability and for ensuring that the defects are handled correctly. Defects change state on a few occasions and for different reasons. They have their own life-cycle.

The life-cycle of a defect loosely follows the states below:

Submit defect -> Assigned for fixing -> Fixed code -> Passed Test -> Closed.

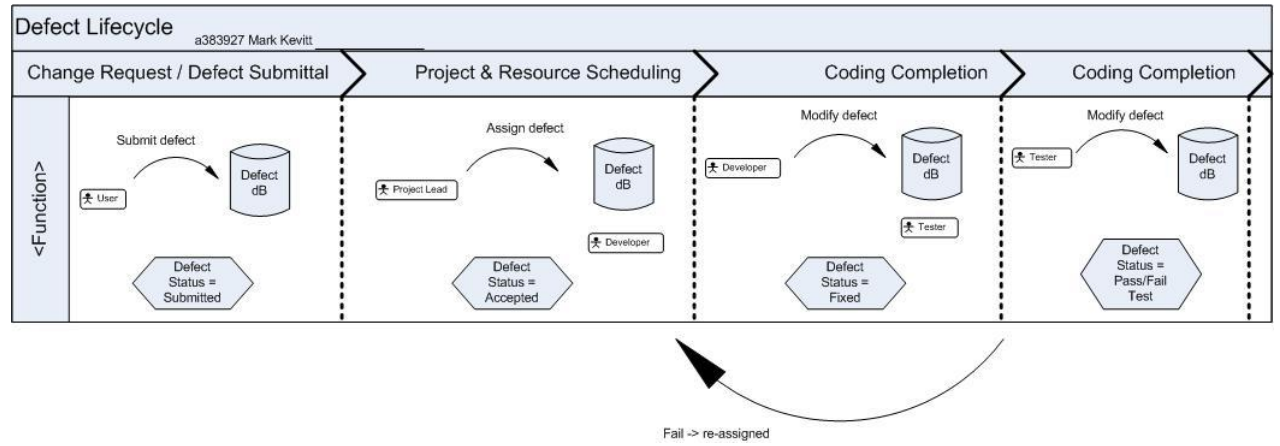


Fig 3.7 Defect Lifecycle

To aid with root cause analysis, when defects are resolved or closed they should be given a category that describes the root cause of the defect. This root cause could be one of a few items:

Root cause	Resolution
Works as Intended:	Not a defect, misunderstanding of the system by the tester.
Code Change	A Code change was required to correct the defect.
Training Required	Customer requires training to fully understand correct use of the system.
Data related	Data anomaly caused the defect.
New Requirement:	A change request for an enhancement or new feature.
Documentation	The current documentation was erroneous, leading for the defect to be created.

Fig 3.8 root cause and resolution

Integrating testing into development life-cycle

Testing should not be considered as a separate process to that of development. It should be an integral part of development, if not then testing will start later than it should and defects will be discovered much later in a product's development. This will have the double-sided effect of having more costly corrections and tardy release of a product.

“Testing must be integrated into the systems development methodology. Considered as a separate function, it may not receive the appropriate resources and commitment. Testing as an integrated function, however prevents development from proceeding without testing” (William E. Lewis, 2004).

There are two fundamental parts to this process.

1. The testing steps must be selected and integrated into the development methodology. For each development stage there must be a corresponding testing stage. This may mean additional tasks for the developers at the respective stage. “The testing steps and tasks are integrated into the systems development methodology through addition or modification of tasks for developmental personnel to perform during the creation of an application system” (William E. Lewis, 2004).
2. Defects must be recorded during the development stages (analysis, design, coding, etc) as they are discovered. This is for the benefit of analysing where defects occur and how to improve their detection. “The test manager must be able to capture information about the problems or defects that occur; without this information, it is difficult to improve testing” (William E. Lewis, 2004).

The steps necessary in integrating testing into development:

- There needs to be a team with understanding of both the testing and development process.
- The development team explain their perspective for each of the stages
- Identify the testing tasks that need to be performed for each development stage
- Establish the overlap between testing and development for each stage
- Modify the development methodology for the new tasks that are required
- Modify the testing methodology for the tasks that are required
- Incorporate and document the defect management process
- Train both teams for the new integrated development methodology

Testing / development of offshore projects

On the occasion that a company requires additional resources to meet the needs of a project team but do not have sufficient resources to do so, it is often more viable and less expensive to outsource the work to an external company. If the external company is located on foreign shores the term is frequently referred to as offshore.

The principal motive for offshore development or testing is financial. There are additional operational overheads associated with offshore projects since the teams conducting the work are working to a larger degree independently of each other.

Once the business proposal and costs have been agreed and the contract signed, in effect the project has begun. It would be good practice for the technical teams to establish a contract in addition to that of the business contract so that each technical team knows what exactly is expected of them and what they can expect from the other team. A good way of doing this is to conduct an audit of each other. To learn each teams local operation is advantageous. This activity has the benefit of learning the process for the other team and also for the identification of commonality and the building of bridges between both parties. A good source of such agreement between both sides would be a project quality plan.

Project Quality Plan

1. The requirements must be agreed before the project can begin
2. The roles and responsibilities of each team participant is defined
3. The quality standards that must be adhered to for each deliverable is stated
4. The methodologies for each team must be explained and agreed
5. The milestones during the project are outlined in a project schedule
6. The content of each deliverable must be clearly stated
7. The entrance and exit criteria for each stage must be defined
8. There should be business knowledge transfer during the discussion of the project and when the project teams are assembled, including the supply of all relevant documentation.
9. Establish the development and testing environment for the offshore team (The hardware, software, tools, licences and data relevant to the project must be agreed upon and if necessary transferred to the offshore team)

These quality principles would be defined after the documentation of each party has been read. It is the precursor to a project plan and perhaps the test plan. When this quality plan has been agreed the project manager draws up a schedule for the contract. The project plan would come under the auspices of QA rather than testing, QA is discussed in detail in the next chapter.

3.4 Summary

In this chapter, the goal for software testing success – error detection was explained and how this goal differs to that of software development. The basis for detecting errors in the program lies with the creation and execution of test cases. The test cases are derived in different manners dependent on the visibility of the internal structure of the application under test (AUT), hence white, black and grey box testing. These three different testing principles were explored and how each type would be typically used.

The system testing needs on the complete system was mentioned and why depending on the objectives of the system that further specific testing is necessary. Two topical types were explained and why they would be required. The combination of the test types and stages are assembled into a test process.

The four elements to the test process were examined, the planning of tests and their subsequent execution, the management of the defects that the test execution detects and finally the release of the build to the end user. The testing process can be independent to the development effort but the benefits and activities to integrate it with the development process were discussed. The last topic that was mentioned was in relation to conducting test process or development off shore. The main topic of this chapter was in relation to the test process itself and its components.

“Software testing is a popular risk management strategy. It is used to verify that functional requirements were met. The elimination of this approach, however, is that by the time testing occurs it is too late to build quality into the product” (William E. Lewis, 2004).

The testing of a program alone does not guarantee that it will be error free or that it will meet with its intended requirements, as these requirements are susceptible to human error. It is a significant leap in the direction of being error free. A step further toward being error free is to test or evaluate each stage the development effort to ensure that the delivered program meets with its intended purpose and that it does this

in a manner that bears well on its creators. A high quality program can not be achieved by testing alone; further effort must be made to achieve this. The quality of the completed program reflects on the efforts of all project parties, to this end, the quality assurance of the program and indeed the part of each party in its creation must be placed under scrutiny.

4 Chapter Four – Quality Assurance

4.1 Complications with software and its quality assurance

Quality Assurance has its roots in assuring the quality of a manufactured physical product; this is achieved by inspecting the product and evaluating its quality near its completion or at various stages of production. Software however is not as tangible as products that are more physical. Typically, a software product is its functionality and not its use. There is no physical software product to evaluate; there is code and not always accompanying documentation. This “invisible” nature of software adds to the complications of assessing its quality. “Industrial products are visible, software products are invisible. Most of the defects in an industrial product can be detected during the manufacturing process, however defects in software products are invisible, as in the fact that parts of a software package may be absent from the beginning” (Daniel Galin, 2004)

There are further complications with assessing software quality; this is attributed to its inherent complexity. Software systems have grown from standalone systems on a single server to globally networked servers spanning multiple countries, and multiple servers. There are now multiple layers to software, where each layer must interface with the software layer above and that below before interfacing with other external systems.

Software may be developed by a team of people who carry out specific roles; the roles are played out during different stages of development. The teamwork driven development life-cycle is open to a multitude of problems, particularly because of the inter-dependence of people in the life-cycle. These problems come in many forms, such as how well the team gel together. Poor relationships between individual team members affect the productivity and creativity of the team. The experience of the team can also have implications where experienced members are supporting inexperienced members. If a project team member departs during the middle of the life-cycle, the consequences of this departure can impact on the success of the project.

These complications are present in other team orientated projects, but the invisible and intangible nature of the software product compounds this further.

The software development team is also affected by external factors such as the customer's documented requirements and how detailed and accurate they represent the actual requirements. The schedule and budget allocated to the project will also have an effect on the quality of the software. After a project has been completed and installed in its target environment, the system must then be maintained for the duration of its lifespan, the ease with which these changes are conducted successfully can affect the quality of the system.

Software Quality is open to discussion and differing authors on the topic have different views on the source of the quality attributes. Crosby (1979 quoted in Daniel Galin, 2004, p.24) defines quality as both the 'conformance to requirements' and 'non-conformance implies defects'. Juran and Gryna (1970 quoted in Daniel Galin, 2004, p.24) refer to software quality as 'fitness for use' and 'customers impression' and later 'freedom of deficiencies'. A third view is that of Pressman (2000 quoted in Daniel Galin, 2004, p.25) who states that there are three requirements for software quality, namely 'Specific functional requirements', 'adhering to quality standards in the contract' and lastly 'Good software engineering practices'.

Each of the three views on software quality has alluded to ways of measuring the quality of the developed software. The whole process of developing the software is best described as 'Software Engineering' and the measurement of the quality of the software is done during the 'Quality Assurance' activity. The software engineering includes the development of the software from customer requirements to a delivered software product, the product quality can then be described in terms of the number of defects that arise in the software. The Software Quality Assurance (SQA) domain lies in the quality management of software during the software engineering development process, SQA defines and measures the inputs and outputs of the development processes and quantifies the quality of the software in terms of defects. In order to measure the software quality it is advantageous to know what to measure.

4.1.1 Factors that impact Software Quality

McCall (1977 quoted in Daniel Galin, 2004, p.37) has identified three different categories of factors that software quality can come under. The factors are spread over the lifespan of the application and not only its original development.

The first set of factors is associated with the original operation of the software product by a user. The second set of factors is directed towards the revision of the product from an existing product to new or enhanced product and how the quality of the original design and code allows for this revision. The last set of factors is concerned with the transition of the product to another target environment, such as a new data base or operating system.

The factors are outlined in each of the tables below:

Quality Factors for new software development		
Product operational	Product revision	Product transition
• <i>Correctness</i>	• <i>Maintainability</i>	• <i>Portability</i>
• <i>Reliability</i>	• <i>Flexibility</i>	• <i>Re-usability</i>
• <i>Efficiency</i>	• <i>Testability</i>	• <i>Interoperability</i>
• <i>Integrity</i>		
• <i>Usability</i>		

Fig 4.1 Mc Calls Quality factors for new software development

Examples of each of these Quality factors are mentioned briefly overleaf:

Mc Calls Software Quality Factors

Product operational	
Correctness	<p>The accuracy of the outputs.</p> <p>Completeness of the output (If 20 chars are input, then 20 chars should be displayed).</p> <p>Timeliness of output (< 3 seconds response time for on-line trading).</p>
Reliability	<p>Mean time between failure (MTBF) - the average time between a number of transactions that one can fail.</p> <p>The allowable downtime for a server is 5 minutes per year of operation</p>
Efficiency	<p>The number of resources required to perform all functions of the software within predefined time frames.</p> <p>The response time for each transaction must be less than 3 seconds.</p>
Data Integrity	<p>Security of the system. Prevention of un-authorised users to gain access.</p> <p>Prevention of critical data being transmitted over the network. Encryption is used where necessary.</p>
Usability	<p>Ease of use for unfamiliar users to become acquainted with the UI.</p> <p>Ease of navigation.</p> <p>Intuitive to use, the user can very quickly learn how to use the application.</p>

Fig 4.2 Mc Calls Product Operational Quality Factor examples

Product revision

“According to Mc Call model of software quality factors, three quality factors comprise the product revision category. These factors deal with those requirements that affect the complete range of software maintenance activities: corrective maintenance, adaptive maintenance and perfective maintenance” (Daniel Galin, 2004)

Product revision	
Maintainability	This is determined by the amount maintenance effort that will be needed by users and development teams to maintain the functionality of existing functionality and to add in new functionality. This is an indication of the modularity of the structure of the software.
Flexibility	The capability and efforts required to support adaptive maintenance on the system. The easier it is to adapt the software to maintenance activities the more flexible it is.
Testability	The ease with which the QA can be conducted on the system. The amount of built in diagnostic support to facilitate the testing of the system for end users, testers and system administrators.

Fig 4.3 Mc Calls Product Revision Quality Factor examples

Product transition	
Portability	Portability deals with a change in environment, e.g. Hardware or Operating system.
Re usability	Re usability requirements deal with the ease of the use of existing software modules in another product or system.
Interoperability	Interoperability requirements focus on the use of the product with other systems.

Fig 4.4 Mc Calls Product Transition Quality Factor examples

Where McCall’s quality factors are used to calculate the quality of the product and particularly the code and documentation they do not take into consideration other quality factors such as the project size, complexity or the team of developers and testers themselves. Other factors that influence greatly the quality of the software produced include the following:

Project factors

- Magnitude of the project
- Technical complexity and difficulty
- Extent of reuse of software components
- Severity of failure outcomes if the project fails

Team factors

- Professional qualifications of the team members
- Team acquaintance with the project and its experience of the subject domain
- Availability of staff members who can support the team
- Familiarity within the team members, the ratio of new people versus existing team members

4.2 Software Quality Assurance

- A planned and systematic pattern of all actions necessary to provide adequate confidence that a software work product conforms to established technical requirements.
- A set of activities designed to evaluate the process by which software work products are developed and/or maintained.

(IEEE quoted in Daniel Galin, 2004) also SEI Carnegie Mello University Glossary of terms for CMM Key practices.

For the purpose of this thesis Software quality assurance (SQA) is considered a process for the measurement of deliverables and activities during each stage of the development lifecycle. The objective of SQA is to quantify the quality of the products and the activities giving rise to them and also to guide a quality improvement effort. It is advantageous to integrate it into the software development process. SQA should also take into consideration the maintenance of a product, the technical solution, product budget and scope. Quality assurance differs from quality control in that quality control is a set of activities designed to evaluate the quality of a developed or manufactured product. The evaluation is conducted during or after the production of the product. Quality assurance however reduces the cost of guaranteeing quality by a variety of activities performed throughout the development and manufacturing process.

For the purpose of this thesis I will focus on the following aspects to SQA. Each SQA activity that I discuss is modular; the SQA activities take place at each developmental stage of the development lifecycle. The stages are categorised into areas for requirements capture, system design and coding and testing and finally release.

1. **Verification** – The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
2. **Validation** – The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specific requirements
3. **Qualification** – The process used to determine whether a system or component is suitable for operational use.

During the analysis, design and coding stages of product development the outputs of each stage need to be measured, monitored and managed so that each output can be verified against its predefined exit criteria. When the final product has completed the coding and integration stages it must be validated against the original user requirements and signed off by senior team members as passed validation testing. At each stage of this product development the efforts during the development must be improved upon where possible in order to cut costs and remain competitive.

This is not an easy task when what is being produced is a program, which in itself is intangible. This is where the complications of software quality assurance lie.

4.2.1 Verification versus Validation

Verification originated in the aerospace industry during the design of systems. There are two criteria:

1. The software must perform all intended functions
2. The software must not perform any function itself or in combination with other functions that can degrade the performance of the system.

An effective verification effort must show that all requirements have been carried out correctly, this is done by testing the requirements against the product during delivery. These tests can be reexecuted to achieve the same results should the system be changed at a later date.

Verification is showing that a product meet its specified requirements at predefined milestones during the development life-cycle. Validation checks that the system meets the customer's requirements at the completion of the development life cycle. An example system of verification versus validation is depicted below:

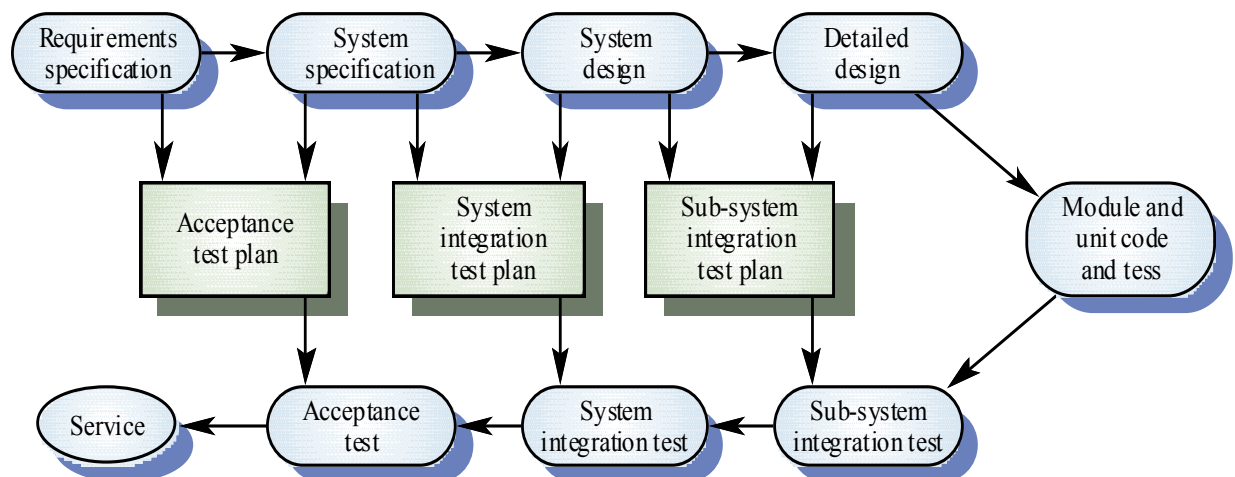


Fig 4.5 V-model of verification versus validation

4.3 Software Quality measurement

“Satisfaction with the overall quality of the product and its specific dimensions is usually obtained through various methods of customer surveys. For example the specific parameters of customer satisfaction in software monitored by IBM include the CUPRIMDSO categories (Capability, usability, performance, reliability, installability, maintainability, documentation, service and overall) for Hewlett-Packard they are FURPS (functionability, usability, reliability, performance and service” (Stephen H. Kan 2003, p.98)

The quality of the software that is produced in each process or model is described in terms of the number of defects that are created. Typically the most common metric for defects is the number of defects per thousand lines of code, or there is another slightly different metric for the defects rate in terms of function points analysis (FPA) abbreviated to (FP).

Defect rate = Sum of Defects / KLOC

Defect rate = Sum of Defects / FP

A line of code is derived from the physical lines of code that the developers write that constitutes the input to the compiled software. A function can be defined as a collection of executable statements that performs a certain task, together with declarations of the formal parameters and local variables manipulated by those statements (Conte et al., 1986). The number of function points refers to the number of functions that are in the software code.

A more recent version of FPA – Mark II is used “to measure the functional size of any software application that can be described in terms of logical transactions, each comprising an input, process and output component, it is a method for the quantitative analysis and measurement of information processing applications. It quantifies the information processing requirements specified by the user to provide a figure that expresses a size of the resulting software product. This size is suitable for the purposes of performance measurement and estimating in relation to the activity

associated with the software product” (United Kingdom Software Metrics Association, 1998, internet) The number of function points is derived by multiplying the function count (FC) by the value adjustment factor (VAF). The FC is derived by summing the grand total of the number of each of the five weighting factors multiplied by the number of components.

FP = FC * VAF	
$FC = \sum_{i=1..n} w_i * x_i$ <p>w is the weighting factors and x is the number of components.</p>	$VAF = 0.65 + 0.01 * \sum c_i$ <p>c is the total of the scores of characteristics, and i = 1 to 14.</p>
<p>For the Function Count (FC) there are five weighting factors:</p> <ol style="list-style-type: none"> 1. 4* Number of external inputs 2. 5* number of external outputs 3. 10* Number of logical files 4. 7* Number of external interface files 5. 4* Number of external inquiries 	<p>The VAF is an assessment on the impact of 14 general system characteristics in terms of their likely effect on the application. It is scaled in the range of zero to five. The 14 general characteristics are:</p> <ol style="list-style-type: none"> 1. Data Communications 2. Distributed functions 3. Performance 4. Heavily used configurations 5. Transaction rate 6. Online data entry 7. End user efficiency 8. Online update 9. Complex processing 10. Re-usability 11. Installation ease 12. Operational ease 13. Multiple sites 14. Facilitation of change

Defects can then be expressed in terms of KLOC's or FP's. The defect rate is defined as the number of defects per function point or thousand lines of code.

4.3.1 Software Defects

With respect to the software, there are three classifications of software defects or bugs as they are more commonly referred as:

1. Software Error
2. Software Fault
3. Software Failure

A **software error** occurs during the development of the software. This error can be in the form of a grammatical error, a logical error where the outcome of a sequence of executions will not result in what was intended or a misinterpretation of the user requirements in the actual written code. It may be in the form of user documentation not matching the software applications operation. An error may or may not be detected during the coding or testing of the program before it is released to a customer.

A **software fault** occurs as a result of an error that remains in the executing program. Not all faults however are detected and the software may continue executing without any obvious problems. There are cases where software faults go undetected for many years of a programs existence.

A **software failure** is a fault that results in a detectable problem; hence it is referred to as a failure. A failure would cause the application to malfunction in an obvious manner that warrants the attention of system maintenance.

4.3.2 Classification of Software Errors

Software errors can be categorised according to the different stages in which they occur in the development life-cycle. “Software errors are the cause of poor software quality, it is important to investigate the causes of these errors in order to prevent them. A software error can be a ‘code error’, a ‘procedure error’, a ‘documentation error’, or a ‘software data error’. (Daniel Galin, 2004). It should be emphasised that the causes of all these errors are human, made by systems analysts, programmers software testers, documentation experts, managers and sometimes clients and their representatives. The causes of software errors can be classified further according to the stages of the software development process.

For each development process stage a number of possible errors are mentioned:

Development Stage	Possible errors
<p>Business Requirements:</p> <p>The errors are caused by human interaction problems. At this stage it is the Business analyst and customer involved in capturing the business requirements.</p>	<ul style="list-style-type: none"> • Faulty definition of requirements • Absence of important requirements • Inclusion of unnecessary requirements
<p>Systems Analysis:</p> <p>Analysis of the system based on the business requirements by the lead developers. The interpretation of the requirements is a risk of causing an error</p>	<ul style="list-style-type: none"> • Misunderstanding of original requirements • Misunderstanding of change requests • Misunderstanding of reported problems
<p>Design:</p> <p>During the system design stage deviations from the requirements are possible where errors can be made.</p>	<ul style="list-style-type: none"> • Software reuse that is not 100% compatible • Leaving out some requirements due to time constraints • Deviations from requirements as a result of creativity
<p>Coding of modules:</p> <p>During the coding the developers may make a</p>	<ul style="list-style-type: none"> • Algorithms • Sequence of component execution • Boundary conditions

Development Stage	Possible errors
number of code errors.	<ul style="list-style-type: none"> • Error handling • Interfacing • Improper use of the software language • Poor programming practice • Unit Testing
<p>Coding integration:</p> <p>When integrating the different modules together, a number of errors can occur.</p>	<ul style="list-style-type: none"> • Integration problems when integrating the code modules • Overly complex code • Interfacing problems • Maintenance problems • Interface testing problems
<p>Testing:</p> <p>There are a number of errors that the Test Engineer can make during the testing stage:</p>	<ul style="list-style-type: none"> • incomplete test plans • failure to document detected errors • failure to promptly correct detected errors due to insufficient defect descriptions • incomplete testing due to time pressure
Delivery and documentation:	<ul style="list-style-type: none"> • Design documentation not kept up to date • User manual errors – out of date descriptions for use • Delivery of incomplete documentation for maintenance teams

Fig 4.6 Classification of software errors

The number of defects that enter in the project (termed defect injection, see fig 4.7) increases with the continuation of the phases of software development. “Phase defect removal effectiveness and related metrics associated with effectiveness analyses are useful for quality planning and quality management. These measurements clearly indicate which phase of the development process we should focus on for improvement”. (Stephen H. Kan 2003, p.172)

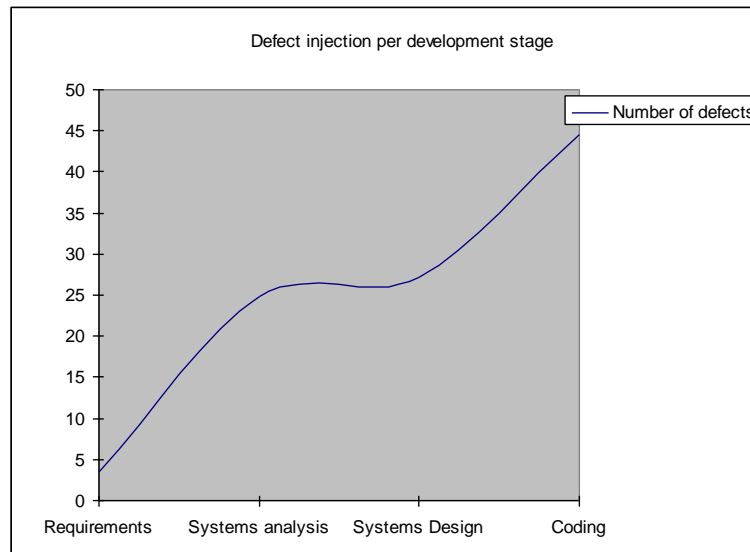


Fig 4.7 Defect injection Rate per development stage

4.4 Structure of Software Quality Assurance (SQA)

In order to derive a plan for SQA, we must revisit the elements of software quality assurance. The fundamentals of SQA deal with a planned activity to evaluate the development process during its progress. This plan or architecture must be placed around the entry to and the output from each stage of the development effort.

If the location and cause of the software defects or errors are taken into consideration during the software development, then there is a starting point for assuring the quality of each stage. These defects can also be considered in relation to *the factors that affect the software quality*. The classification of the causes of the defects can be addressed by SQA.

These combined factors that concern software quality, are the building blocks of an SQA Architecture as per figure 4.5 (V-model of verification versus validation). SQA is a continuously evolving entity with an emphasis on improving. There are three parts to this architecture; they are listed below in figure 4.8.

The Architecture of SQA

SQA Component	Activities
1. <i>Planning from the project initiation and planning stage</i>	<ul style="list-style-type: none">• Review and plan the project in its entirety• Create the QA plan
2. <i>Management of the Project life-cycle activities and components</i>	<ul style="list-style-type: none">• Create a defect removal and defect injection prevention
3. <i>Refactoring the Management of all SQA components</i>	<ul style="list-style-type: none">• Instigate Software Quality improvement

Fig 4.8 Structure of SQA

4.4.1 Planning from the project initiation and project planning stage

Projects that are carried out ‘in house’ are more susceptible to failure than projects which go under the more formal external contract route. For this reason the schedule and budget failures are accompanied by lower than acceptable software quality, this is largely due to a more casual attitude to meet deadlines. Contract review can alleviate this by ensuring that the correct measures are put in place for the project. Following the contract review, the project plans and schedule should be documented. Any risks that are envisaged at this stage should also be documented with a probability of occurrence and a mitigation plan identified should the risk occur.

Contract Review

Purpose of the contract review:

- Clarification and documentation of customers requirements
- Formal aspects of the business relationship and identification of responsibilities
- Communication hierarchy, deliverables, acceptance criteria, formal phase approval process, design and test follow up process, change request procedure
- Estimation of project resources and timetable
- Estimation of company’s exposure with respect to the project
- Estimation of the customer’s capacity to meet commitments
- Definition of intellectual property rights

The ‘failure to review’ can leave the project open to errors in relation to inadequate definition of requirements, poor estimates of required resources, overrun of the schedule or budget which impacts the team effort and hence quality. To alleviate this, output from the review can be used as an input to the documentation of plans for both development and quality assurance

Documentation of Development and Quality plans

“Development and quality plans are major elements needed for project compliance with 9000.3 standards. It is also an important element in the Capability Maturity Model (CMM) for assessment of software development organisation maturity” (Daniel Galin, 2004). For a project to be successful, a number of project plans need to be prepared. The following tasks need to be performed following the contract review:

- Scheduling of development activities.
- Estimation of resources and budget.
- Recruitment and allocating of resources
- Identifying and risk assessment
- Providing reporting structure for project control

In addition to the elements mentioned in Galin 2004, a Development plan ideally would contain the following elements:

Elements of a development plan	
1. Project products	<ul style="list-style-type: none"> • Design documents with dates of completion • Set of deliverables • Software products with completion and installation site. • Training tasks
2. Project interfaces	<ul style="list-style-type: none"> • Interfaces with existing SW packages • Interfaces with other software (dev teams) • Interfaces with HW
3. Project methodology & development tools	<ul style="list-style-type: none"> • UML • Case Tools
4. Software development standards and procedures	<ul style="list-style-type: none"> • e.g. Coding conventions
5. Mapping of project phases	<ul style="list-style-type: none"> • Estimate of phase duration • Logical sequence of phase completion • Estimate of external resources required. • Presentation using Gantt chart e.g. MS project, critical path analysis. Start time, end time &

Elements of a development plan	
	dependencies.
6. Project milestones	<ul style="list-style-type: none"> • Completion dates
7. Staff Organisation	<ul style="list-style-type: none"> • Roles and responsibilities
8. Development facilities	<ul style="list-style-type: none"> • Tools • Equipment
9. Development risks	<ul style="list-style-type: none"> • Language • Tool experience • Staff shortages • Independence of external suppliers
10. Risk Management Action	<ul style="list-style-type: none"> • Risk identification • Risk evaluation • Mitigation planning • Risk weighting
11. Control methods	<ul style="list-style-type: none"> • Reports • Meetings
12. Project cost estimation	<ul style="list-style-type: none"> • Contract • Schedule • Estimates

Fig 4.9 Elements of a development plan

Risk Management is a contributing factor to software quality should the risks materialise. It is worthwhile for SQA to independently evaluate that risk analysis and planning has been performed. “Identification of software risk items (SRI) should begin with the actual start of the project (pre-software stage) and be repeated periodically throughout the project until its completion”. (Daniel Galin, 2004)

The evaluation of the identified SRI should be conducted for contingency plans to be put in place. A list of the SRI’s should be compiled and a priority assigned to each risk in terms of determining the risk exposure.

$$\text{Risk exposure} = \text{Probability of materializing} * \text{Estimate damage}$$

A typical quality plan should contain the following items:

Elements of the Quality plan	
1. List of Quality Goals	<ul style="list-style-type: none"> • Quantitative – error severities • Qualitative measurements (Downtime, response time, throughput etc)
2. Review Activities	<ul style="list-style-type: none"> • Design review, test case reviews, etc
3. Software tests	<ul style="list-style-type: none"> • Test strategy, plan, test design, environment etc
4. Acceptance tests	<ul style="list-style-type: none"> • Test strategy, plan, test design, environment etc
5. Configuration management	<ul style="list-style-type: none"> • Change control, version control etc.

Fig 4.10 Elements of a Quality Plan

4.4.2 Management of the Project life-cycle activities and components

Software Quality Assurance Defect Removal

Considering that there are several factors that affect software quality there are a number of activities that can be followed to improve the development stages in terms of software quality. The activities are discussed below.

- 1. Reviews**
- 2. Inspections**
- 3. Walk through**
- 4. Testing**
- 5. Configuration management**

“An inspection and walkthrough is an improvement over the desk-checking process (the process of a programmer reading his or her own program before testing it). Inspections and walkthroughs are more effective, again because people other than the programs author are involved in the process. These methods generally are effective in finding from 30 to 70% of the logic-design and coding errors in typical programs” (Glenford J. Myers, 2004).

Procedural order and teamwork lie at the heart of formal design reviews, inspections or walk-through. Each participant is expected to emphasise his or her area of expertise. The knowledge that the work item will be reviewed stimulates the team to work to their upper end of productivity.

For different stages of the development process, there are different defects that get injected into the software. The rate of defect injection differs for each stage of development. The QA activities must match the defect injection rate and type to be effective at their removal. Fig 4.11 demonstrates the distribution of defect injection for each of the four phases of the development process. Fig 4.12 identifies the effectiveness at defect removal by QA activity and development phase. Lastly the cost

associated with the QA activities are listed in Fig 4.13

“Defect origins (the phase in which defects were introduced) are distributed throughout the development process, from the projects initiation to its completion” A characteristic distribution of software defect origins based on Boehm (1981) and Jones (1996), is shown below”. (Daniel Galin, 2004)

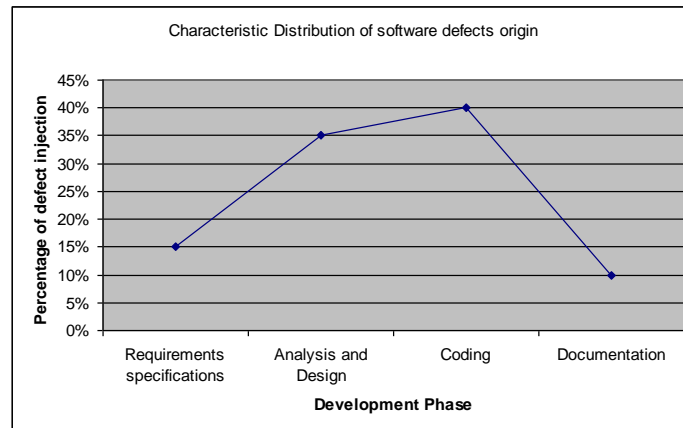


Fig 4.11 Characteristic Distribution of software defects origin (Daniel Galin, 2004)

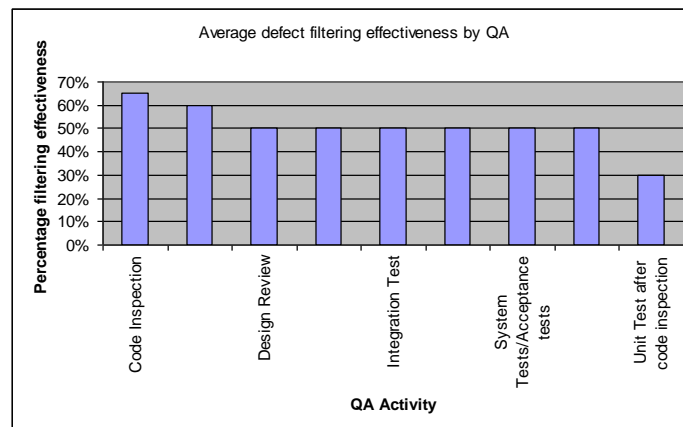


Fig 4.12 Average defect filtering effectiveness by QA (Daniel Galin, 2004)

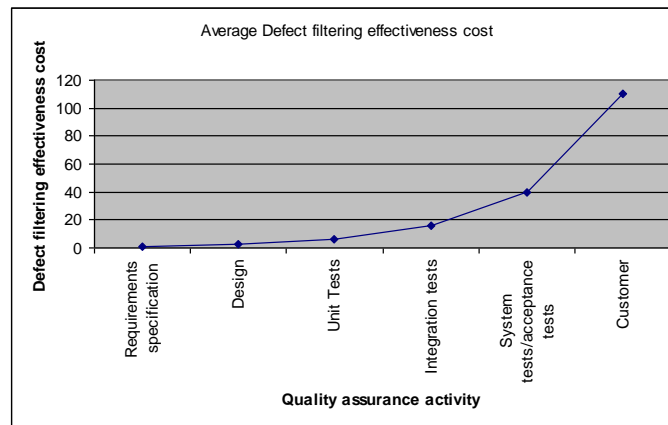


Fig 4.13 Representative average relative defect removal costs (Daniel Galin, 2004)

Reviews

The foundation of reviews is based on the human disposition to produce errors. The author of a document or code is unlikely to discover their own errors irrespective of the number of checks that they conduct. It is necessary for independent peers, experts, superiors or customers to conduct a review of the artefact in question.

“Only others - those having different experiences and points of view, yet not directly involved in creating the document are capable of reviewing the product and detecting the errors unnoticed by the development team”. (Daniel Galin, 2004. pp. 150)

These reviews provide early detection and prevent passing of design and analysis errors downstream. It can also detect defects in the coding phase.

Direct objectives of reviews:	Indirect objectives of reviews :
Detect analysis and design errors	Exchange of professional knowledge – tools, techniques etc
Identify new risks likely to affect the completion of the project	Record analysis and design errors for future references
Approval of the work under review	Collaboration between teams

Fig 4.14 Objectives of reviews (Daniel Galin, 2004)

Formal design review:

Without the approval of this review the project may not continue on to the next stage. “Because the appointment of an appropriate review leader is a major factor affecting the DR’s success, certain characteristics are to be looked for in a candidate for this position”.

1. *Review leader should have experience in development projects of this type.*
2. *Seniority at a level similar to the project leader*
3. *Have a good relationship with the project lead and the team.*
4. *A position external to the project team.*

(Daniel Galin, 2004)

Galín also mentions the review process in detail and explains what is required for the successful execution of design reviews.

The design review (DR) process consists of the following items:

- Preparation
- DR Session
- DR Report

Inspections

Inspections are more formal than reviews.

1. Inspections should contain professional participants who are acquainted with the language and technology being used on the project.
2. An architect who is responsible for the analysis and design of the system under review.
3. A coder who is familiar with the code language and who can spot errors.
4. A Tester who can give a QA perspective who can spot defects that would normally be discovered at testing.

Walk-through

“The code walkthrough, like the inspection, is a set of procedures and error-detection techniques for group code reading. It shares much in common with the inspection process, but the procedures are slightly different, and a different error-detection technique is employed” (Glenford J. Myers, 2004). It is less formal than reviews and should contain professional participants who are acquainted with the language and technology being used on the project:

1. A standard enforcer who is familiar with the coding standards and procedures.
2. A maintenance expert who can focus on maintainability, testability, performance and other areas of concern for maintainability.
3. A user representative who can focus on the user’s perspective.

It is advantageous to have a presenter who is not the author so that any anomalies can not be glossed over. It is also beneficial to have a scribe to take notes.

During the walkthrough not all work is mandatory for review. What should and should not be subjected to a walkthrough is listed in fig 4.15 below:

<i>In</i>	<i>Out</i>
Complicated logic	Straightforward logic
Critical sections	Familiar sections
New sections	Low severity sections
Inexperienced developers	Reused code / sections

Fig 4.15 Comparison of items subjected to a walkthrough

Testing Process

Testing has been discussed in great length in chapter three. It is worth mentioning that the testing must be planned with the project risks and quality attributes in mind. These will have been identified in the quality plan.

Software Configuration management.

Software configuration management (SCM) is concerned with labelling, tracking, and controlling changes in the software elements of a system. The purpose of software configuration management is to control code and its associated documentation so that final code and its descriptions are consistent and represent those items that were actually reviewed and tested.

Software configuration management identifies a system configuration in order to systematically control changes, maintain integrity, and enforce traceability of the configuration throughout its life-cycle.

Components to be controlled include plans, analysis, design documentation, source code, executables, test plans, test cases and reports. The SCM process typically consists of five elements:

1. Software component identification
2. Software version control
3. Configuration building
4. Change control
5. Templates and Checklists

Component Identification

Identification of components that make up a deliverable at each point in its development. A component would typically consist of a certain amount of code that collectively contains a number of functionality. Each component should be identified by a meaningful name and version number, such that new revisions contain enhanced functionality. The ability to roll back to previous revisions should be available.

Version control

This is the organised process to manage the changes in the software components and their relationships. This creates the ability to support parallel component development and maintenance. A component is identified and labeled to differentiate it from all other software versions and components.

Change Control:

“Change control is the process by which a modification to a software component is proposed, evaluated, approved or rejected, scheduled and tracked. Its basic foundation is a change control process, a component status reporting process and an auditing process” (William E. Lewis, 2004. pp. 15 - 16). There should also be an impact analysis conducted to determine the dependencies of components. Change control consists of a change request, an impact analysis, a set of modifications and new components and a method for reliably installing the modifications as a new baseline.

Templates and Checklists

Templates

“A template refers to a format created by units or organisations, to be applied when compiling a report or some other type of document” (Daniel Galin, 2004. p 326). It is a format that is created which is intended to be reproduced several times. The template document can be designed as a starting point for the reproduction of similar documents. The purpose of a template is to facilitate copying with outline generic contents which will act as a prompt to future authors. The templates can be written for every document on projects including, plans, tests and code. Templates will save time on future projects as they represent a part complete new document. Other benefits to templates include training material for new team members

Checklists

“The checklists used by software developers refer to the list of items specifically constructed for each type of document, or a menu of preparations to be completed prior to performing an activity” (Daniel Galin, 2004. p. 329) Checklists serve two purposes, they are a list of items specifically constructed that act as a concise list of items to be verified as complete and also provide a record of items that have been verified as complete. Checklists can be applied to any activity or document to serve as a verification record of completion. The dual benefits to checklists are that they serve as a preparation material for an individual preparing for a review and also as a method for the action and record of a review activity. Checklists should be compiled during review activities and updated wherever necessary to keep apace with change.

4.4.3 Defect Prevention process

“The defect prevention process (DPP) is not itself a software development process. Rather, it is a process to continually improve the development process”. (Stephen H. Kan 2003, p.35) This is a lighter process that is again concentrating on continually improving the software quality output from an arbitrary development process. It is based on the following three steps and is in agreement with Deming’s principles.

1. Analyse defects or errors to trace root causes.
2. Suggest preventative actions to eliminate the defect root causes.
3. Implement the preventative actions

The formal process was first used at IBM Communications Programming Laboratory at research Triangle Park, North Carolina (Jones, 1985 Mays et al 1990). It consists of the following four key elements

1. Causal analysis meetings:

After a development stage the technical people analyse defects for that stage and determine the root cause. The defects are updated with suggested actions by the meeting leader. Career managers do not attend this meeting.

2. Action Team:

The course of action team is responsible for screening, prioritising and implementing the actions to prevent the re-occurrence of the same or similar defects. The team reports back their findings to management.

3. Stage kick-off meeting:

The technical team conducts these meetings at the beginning of each development stage. The emphasis is on the technical aspect of the development process and quality. The topics for discussion include the

process, efficiency, tools and methods. Items such as likely pitfalls are discussed. The meeting has two purposes firstly as a feedback mechanism of the defect prevention process and secondly as a preventative measure.

4. Action tracking and data collection:

A database is used for tracking actions, their status and for communicating the findings to a broader audience.

DPP is a real time process and occurs at each development stage. It is incorporated into every process and sub process. It helps focus the entire team towards defect prevention. It requires the support of management.

IBM's Network Communications Program had a 54% reduction in error injection during development and a 60% reduction in field defects after implementation. IBM in Houston, Texas, developed the space shuttle onboard software control system with DPP and achieved zero defects since the late 1980's. Causal analysis of defects along with actions aimed at eliminating the cause of defects is credited as the key factors in these successes (Mays et al 1990).

DPP can be applied to any development process as long as the defects are recorded, causal analysis can be performed and preventative actions mapped and implemented. In the SEI software process maturity assessment model (Humphrey, 1989) the element of defect prevention is necessary for a process to achieve the highest maturity level – level 5.

It is worth mentioning that there are national awards for quality achievement in countries across the globe. The Malcolm Bridge Assessment National Quality Award is the most prestigious award in the US. The award is given to US company's that excel in quality achievement. In 1992 the European Foundation for Quality management published the European Quality Award. It is similar to the Malcolm Bridge award.

4.4.4 Capturing and analysing defect metrics

Reliability Models

Software reliability models are used to assess software product reliability or to estimate the number of latent defects when it is available to customers. There are two direct benefits for using reliability models:

1. As an objective statement of the quality of the product
2. Resource planning for the maintenance effort

Reliability models typically capture the number of defects per KLOC or the number of defects/FP (Function Points).

Rayleigh Model:

The Rayleigh model is a member of the family of the Weibull distribution. “The Weibull distribution is an extension of the two-parameter exponential distribution to three parameters. This model is quite popular as a life-testing distribution and for many other applications where a skewed distribution is required.” (Bain, J. Lee, 1978). “It is has been used for decades in various fields of engineering for reliability analysis, ranging from the fatigue life of deep-groove ball bearings to electron tube failures and the overflow incidence of rivers. It is one of the three known extreme-value distributions” (Tobias, 1986, quoted in Kan 2003).

Its cumulative distribution function (CDF) and probability density function (PDF) are:

$$\text{CDF: } F(t) = 1 - e^{-(t/c)^m}$$

Cumulative defect arrival pattern

Where m is the shape parameter, c is the scale parameter, and t is time.

$$\text{PDF: } f(t) = \frac{m(t)^{m-1} e^{-(t/c)^m}}{T(c)}$$

PDF = Defect Density rate or defect arrival pattern.

In both formulas m is the shape parameter, c is the scale parameter and t is time.

It has been empirically well established that large software projects follow a life-cycle pattern described by the Rayleigh density curve (Norden, 1963; Putnam 1978). Using this knowledge past projects and current projects can be compared to each other to determine the state of a project at a number of different stages using graphs of each project as a tool for comparison.

In 1984 Gaffney of the IBM Federal Systems Division developed a model based on defect counts at six phases of the development life-cycle; High level design inspections, low level design inspections, code inspections, unit test, integration test and system test. The defect pattern followed the Rayleigh curve. The model can be used to estimate defects, or project size and resource requirements. By validating the model with systems for which defect data are available (including the space shuttle development) Putnam and Myers (1992) found that the total number of defects was within 5% to 10% of the defects predicted from the model.

Curves that peak earlier have smaller areas at the tail, the release phase. A value of 1.8 for the value of m might be best for software. "Three cases of Rayleigh underestimation discussed are from different software development organisations, and the time frame spans sixteen years from 1984 to 2000. We recommend the use of Weibull with $m = 1.8$ in Rayleigh applications when estimation accuracy at the tail end is critical" (Stephen H. Kan 2003, p. 204)

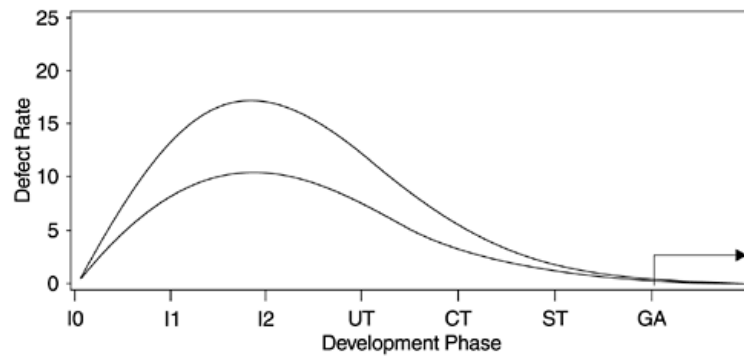


Fig 4.16 Rayleigh model of defect rate versus development phase (Stephen H. Kan 2003 p. 193)

Fig 4.16 indicates two projects with similar time frames but one has a higher defect injection rate and will have a higher defect rate in the field (GA phase).

Exponential Distribution and reliability growth models

In the case of defect distribution, the graph indicates defect arrival or failure patterns during testing and is a good indicator of the products reliability when it is used by customers. They can be classified into two classes:

Fault between failure models (time)

Fault count models (number of faults)

As defects are detected and removed from the software, it is expected that the observed number of failures per unit time will decrease.

The Exponential, Delayed S and Inflection S models

The exponential model is another special case of the Weibull family, with the shape parameter m equal to 1. It is best used for statistical processes that decline monotonically to an asymptote. Its cumulative distribution function (CDF) and probability density function (PDF) are:

$$\text{CDF: } F(t) = 1 - e^{-t/c}$$

$$= 1 - e^{-\lambda t}$$

$$\text{PDF: } f(t) = \frac{1}{c} e^{-(t/c)}$$

$$= \lambda e^{-\lambda t}$$

Where c is the scale parameter, t is time and $\lambda = 1/c$. Applied to software reliability, λ is referred to as the error detection rate or instantaneous failure rate. (Stephen H. Kan 2003 p.208)

The exponential distribution is the simplest and most important distribution in reliability and survival studies. Misra (1983) used the exponential model to estimate the defect- arrival rates for the space shuttles ground system software for NASA. A testing process consists not only of a defect detection process but also a defect isolation process. Because of the time needed for failure analysis, significant delay can occur between the time of the first failure observation and the time of reporting. Yamada et al (1983) offers the delayed S-shaped reliability growth model for such a process, in which the observed growth curve of the cumulative number of detected defects is S shaped. (Based on the non-homogeneous Poisson process)

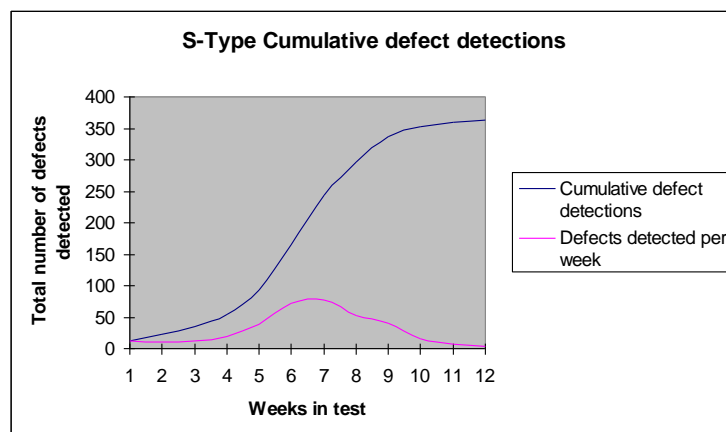


Figure 4.17 S-Type Cumulative defect detections and defects per week

$$M(t) = k [1 - (1 + \lambda t) e^{-\lambda t}]$$

Where t is time, λ is the error detection rate,

k is the total number of defects or the cumulative defect rate. (Stephen H. Kan 2003 p.

215)

4.4.5 Quality Management Models

Rayleigh Curve for Development Quality

The typical use of reliability models for quality management is to predict the end date to testing given a level of defect detection. If the level of detected defects is low then a greater testing effort will be required. The goal of quality management is to shift the peak of the number of defects to the left while also lowering the severity of the peak. The defects for each stage of the development life-cycle are plotted and the resulting curve gives an indication of the phase of greatest defect injection and defect removal.

“The relationship between formal machine-testing defects and field defects, as described by the model (Rayleigh) is congruent with the famous counter intuitive principle in software testing by Myers (1979), which basically states that the more defects found during formal testing the more that remained to be found later on. The reason is that at the late stage of formal testing, error injection of the development process is basically determined. High testing defect rates indicates that the error injection is high, if no extra effort is exerted, more defects will escape to the field”
Stephen H. Kan 2003, p. 236)

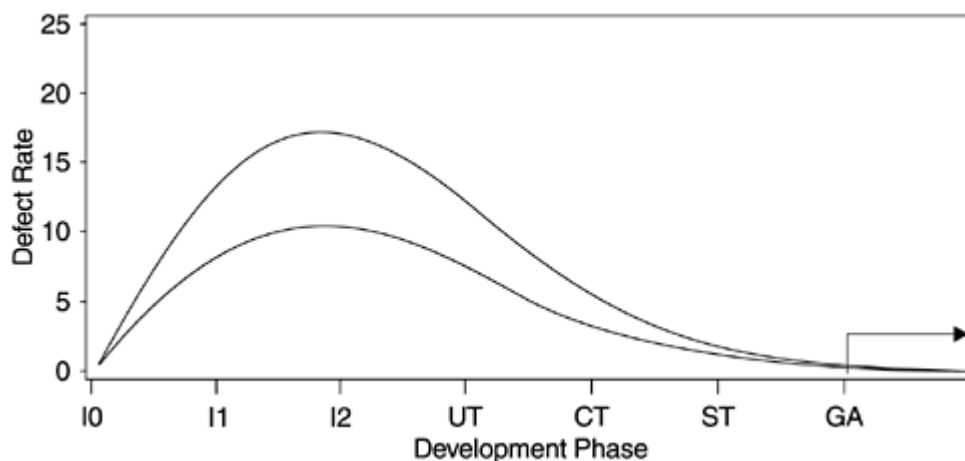


Fig 4.18 Rayleigh model of defect rate versus development phase (Stephen H. Kan 2003 p. 193)

The ultimate target of IBM Rochester's strategy is to achieve the defect injection/removal pattern represented by the lowest curve, one with an error injection rate similar to that of IBM Houston's space shuttle software projects. The development phases are represented by the X-axis and are listed in Fig 4.19.

Review Stage	IBM Rochester abbreviation
1. High level design review	(IO)
2. Low level design review	(I1)
3. Code Inspections	(I2)
4. Unit Test	(UT)
5. Component Test	(CT)
6. System Test	(ST)
7. General availability	(GA)

Fig 4.19 Review stages for concentration

The best curve to have is an early peaking of defects which lowers the total number of defects, and a lower overall curve. However lower actual defect detection and removal could be the result of lower error injection or poor reviews and inspections and in contrast higher defect detection and removal could be the result of higher injection or better reviews. To better gauge which scenario is the case additional metrics are needed. Items such as the number of hours spent in reviews, inspections and testing would assist with identifying which is the case.

The effort / outcome indicator is used for this purpose; the number of hours spent in preparation for and in conducting reviews/inspections is measured with the number of defects per thousand lines of code. This is recorded for each project and can then be used as an indicator for the effectiveness of the defect detection and removal actions.

The purpose of both of these metrics are to determine the in process escape rate and percentage of interface defects. From these metrics the total number of defects found in a phase can be graphed against defects found by previous phases. This graph assists with identifying the effectiveness of the defect removal versus effort.

4.4.6 In process metrics for software testing

Test Progress S curve

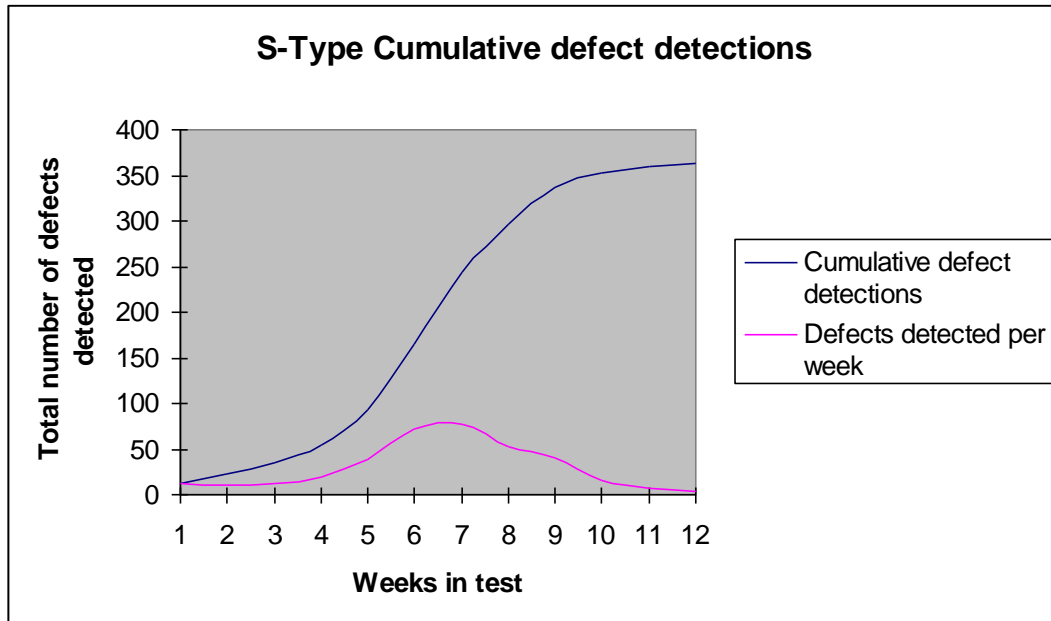
The purpose of this metric is to track test progress and compare it to the plan, and therefore to be able to take action upon early indications that testing activity is falling behind. The Test progress S curve is an accumulative growth curve where the planned number of test cases is measured alongside the actual executed number of test cases. The curve is the accumulated number of planned test cases. It is also beneficial to score the more important test cases, so that there is more meaning to those that are completed. This weighting can be determined at the test plan stage so the implications of any drop off in the curve or test progress is immediately obvious.

Testing Defects Arrival time

The purpose of this model is to model the defects as they are logged in a defect tracking tool. It is important to track the defects over different test phases. Important information can be gleaned from this model such as:

- At what stage do defect peak?
- How does this pattern compare to previous patterns?
- How do they peak?
- Do they decline to a low and stable number?

A positive pattern of defect arrivals is one with higher arrivals earlier. As was mentioned previously this left sided or early peak is an indicator of a good quality process and product. The early peak will also lead to a smaller and earlier tail which indicates less remaining defects in the field for customers. This is visible in a weekly defect arrival pattern. If the curve is plotted as a cumulative defect arrivals curve, the residual number of field defects can be calculated or estimated over time.



6

Fig 4.20 Cumulative defect curve for the arrival of defects over time

Testing Defect Backlog over time

The testing defect backlog is the accumulated difference between defect arrivals and defects that were closed. A large number of outstanding defects during the development cycle will impede test progress.

Test Effort and Defect Outcome model

When measuring the test effort in terms of test cases completed versus those planned, the percentage completion is used as the indicator of effort. When evaluating the outcome of the testing effort, it is best to think along the lines of the number of defects found, the arrival of defects is a good indicator of outcome.

Effort (Test effectiveness) / Outcome (defects found)

	Higher Outcome	Lower Outcome
Better Effort	1. Good / Not Bad	2. Best-Case
Worse Effort	3. Worst-Case	4. Unsure

Fig 4.21 Effectiveness of test effort and outcome (Stephen H. Kan. 2003)

1. This cell indicates good test effort in relation to a number of latent defects found by testing that was injected in the design and code stage.
2. This is the best case scenario where there was less defect injection during the design and code stage, yet the test effort was effective, just less defects were found during testing.
3. This is the worst case scenario where there was a high degree of defect injection in the design and code stages of development and that it took minimal test effort to discover a high number of defects.
4. This is the unsure category where it is inconclusive that the lower number of defects is a result of the testing effort or poor design and coding.

4.5 Refactoring the Management of all SQA components

4.5.1 Software Quality Management

The ground work for a professional approach to assurance of software quality has been established. The complications and factors that act on the software effort have been discussed. The question is how does a quality assurance professional manage an engineered strategy to counter the quality impediments and develop a quality improvement initiative? There are a few different management approaches to answering this question; once again there are similar traits and characteristics to each one.

One such approach is Total Quality Management (TQM); it is derived from a Japanese-style of management where quality assurance was implemented at all levels of the company to improve customer satisfaction. The Principles are management of product quality with customer quality via process improvement and monitoring. The key elements to TQM are:

1. A Customer Focus to achieve total customer satisfaction
2. Process improvement on business and product processes
3. The Human Element to quality, to advocate a company wide quality culture
4. Measurement and analysis of quality metrics to achieve the goal of improved quality
5. There is also a need for Executive leadership in the corporation

To differing degrees TQM has been included in the works of Crosby (1979), Feigenbaum (1961, 1991), Ishikawa (1985) and Juran and Gryna (1970).

Deming (1986) also describes a feedback cycle that optimises a single process for statistical quality improvement. This quality management process involves a Plan-Do-Check and Act philosophy. Experimentation is important with this process and improvement is made based on the analysis of the feedback received.

The Quality Improvement Paradigm (Basili 1985, 1989, Basili and Rombach 1987, 1988, Basili et al 1992) aims at building a continually improving organisation based on evolving goals and an assessment of its status relative to these goals. The approach uses internal assessments and techniques such as Goal/Quality/Metric GQM, model building and Qualitative / Quantitative analysis to improve the product through the process.

The six fundamental steps of the quality improvement paradigm are

1. Characterise the project audits environment
2. Set the goals
3. Choose the appropriate process
4. Execute the process
5. Analyse the data
6. Package the experience for reuse

The Software Engineering Institute (SEI) Capability Maturity model (Humphrey 1989, Radice et al 1985) is a staged process improvement based on the assessment of key process areas until you reach level 5 which represents a continuous process improvement. The improvement is based on organisational and quality management maturity models developed by Likert (1967) and Crosby (1979) respectively.

The goal of this approach is to achieve continuous process improvement via defect prevention, technology innovation and process change management

Based on this approach a five level process maturity model is defined based on repeated assessments of an organisations capability in key areas. Improvement is achieved by action plans for poor process areas. Basic to this approach is the idea that there are key process areas and attending to them will improve your software development.

4.5.2 The SEI Process Capability Maturity model

The Process Capability Maturity Model (CMM) was developed by the SEI at Carnegie-Mellon University. “CMM is a conceptual framework that represents process management of software development. CMM contains five maturity levels or stages” (Joseph Raynus, 1998. p 9)

Level 1: Initial

Level 2: Repeatable

Level 3: Defined

Level 4: Managed

Level 5: Optimising

Level 1: Initial

The characteristics for this stage include chaotic and unpredictable cost, schedule and quality.

Level 2: Repeatable

Characteristics: Intuitive – cost and quality highly variable, reasonable control of schedules, informal and ad hoc methods and procedures. The key process areas (KPA's) to achieve level 2 maturities follow:

- Requirements management
- Software project planning
- Software project tracking and oversight
- Software subcontract management
- Software quality assurance
- Software configuration management

Level 3: Defined

Characteristics: Qualitative – reliable costs and schedules, improving but unpredictable quality performance. The key elements to achieve this level of maturity follow:

- Organisational process improvement
- Organisational process definition
- Training program
- Integrated software management
- Software product engineering
- Intergroup co-ordination
- peer reviews

Level 4: Managed

Characteristics: Qualitative – reasonable statistical control over product quality. The key elements to achieve this level of maturity follow:

- Process measurement and analysis
- Quality Management

Level 5: Optimising

Characteristics: Qualitative basis for continued capital investment in process automation and improvement. The key elements to achieve this highest level of maturity follow:

- Defect prevention
- Technology innovation
- Process change management

The CMMI was developed by integrating practices from four CMMS for software engineering, systems engineering, for integrated product and process development and for acquisition.

4.5.3 Software Process Assessment

“There are two methods suggested by SEI for the software process appraisal: software process assessment and software capability evaluation (SCE). The objective is to evaluate the organisation in the same manner, using CMM’s criteria” (Joseph Raynus, 1998. p33)

SEI developed and published the Capability Maturity Model (CMM) Based Appraisal for Internal Process Improvement (CBA IPI) (Dunaway and Masters, 1986). The data collected for CBA IPI is based on key process areas of CMM as well as non CMM issues.

The standard CMMI Appraisal Method for Process Improvement (SCAMPI, 2001, internet) developed to satisfy CMMI is more stringent than CBA IPI. Both SCAMPI and CBA IPI consist of three phases; planning, assessment and reporting. These phases are outlined below.

Planning:

- Develop the plan
- Prepare and train the team
- Make a brief assessment of participants
- Administer the CMMI appraisal questionnaire
- Examine Questionnaire responses
- Conduct initial document review

Assessment:

1. Conduct the opening meeting
2. Conduct interviews
3. Consolidate information
4. Prepare presentation of draft findings
5. Present draft findings
6. Consolidate, rate and prepare final findings

Reporting:

- Present final findings
- Conduct executive session
- Wrap up assessment

Where the CMM assessments are aimed at CMM derived models, a similar approach can be adopted for assessments in other company's adopted models. A quality assessment is concerned with the quality status of the project rather than the state of process practices although there is likely to be correlation among the two. To achieve an effective quality assessment, the development process, environment and the project plan must be well understood.

4.5.4 Software Quality Auditing

“The concept of auditing is central and is applied at two levels: process and project” (Joc Sanders et al. 1994, p72).

Process level

In Software Quality, Sanders and Curran (1994) discuss that software quality audits are conducted on two levels, that of process and project. Despite the lack of a quality process certification in an organisation there is still a process for developing and maintaining software. This “process may not be formally defined or understood and may even be chaotic, but it is still a process” (Joc Sanders et al. 1994, p72). Other organisations may have a defined standard process which consists of documented standards and procedures that define an environment for developing software, enable discussion of project issues in a common vocabulary, and allow staff to collect and apply experience consistently. Both processes may be audited in a structured manner. Either the SEI or CMM approach to software assessment or a software quality audit may be conducted to assess the process in place, defined or chaotic. Process improvements may be made on the findings of such assessments on a process level.

Project level

“Quality is not imposed on a project, but is controlled and managed from within by project staff. All staff members bear responsibility for the quality of their own work, and the project manager bears overall responsibility for project quality” (Joc Sanders et al. 1994, p72) The project level assessment or audit gives a better understanding and assessment of the process itself since it is not a process but an actual project with people, documentation from all participants and deliverables that provide concrete information pertaining to the project and the process itself.

The purpose of an audit or assessment of a project is twofold: “to determine if activities are being carried out in accordance with the standards and procedures laid down by the quality process and whether those standards and procedures are adequate to ensure the quality of the project in general” (Joc Sanders et al. 1994, p73).

4.6 Summary

In this chapter, the concept of software quality was explained. The factors that affect software quality were discussed, and how these factors are used so that software quality can be assessed. The assessment of software quality is determined by the number of defects in the software. The classification of defects was outlined in terms of faults, failures and errors.

The discipline of software quality assurance was mentioned in relation to the evaluation of software quality and defects. The methods for SQA - verification, validation and qualification were discussed.

The architecture of SQA was described in terms of quality planning from the outset and the assessment and measurement of quality in terms of defects and metrics. The purpose and contents of plans for both development and quality of a maturing organisation were mentioned and their purpose explained. The tools for the prevention of defects were explained and what their benefits are. Lastly quality management was discussed in relation to the activities available for software quality improvement. The final topic of the chapter was in relation to software process assessment and process maturity.

5 Chapter Five – Software Test and Quality Assurance Practice Improvement

5.1 The first steps to test and QA practice improvements

In this chapter the path to testing and QA practice improvement in an Irish small to medium enterprise (SME) is explored, for the purpose of this thesis the identity of the company is withheld. This thesis is focused on the projects from the R&D department and on the improvements to the testing and quality assurance of its products.

I will introduce the industry that the company operates in; I will also describe their products from a software engineering / Research and Development department perspective. I will then outline what quality problems the company faced and lastly outline my proposal to the company to address these problems. The organisation structure of the company is depicted below in figure 5.1.

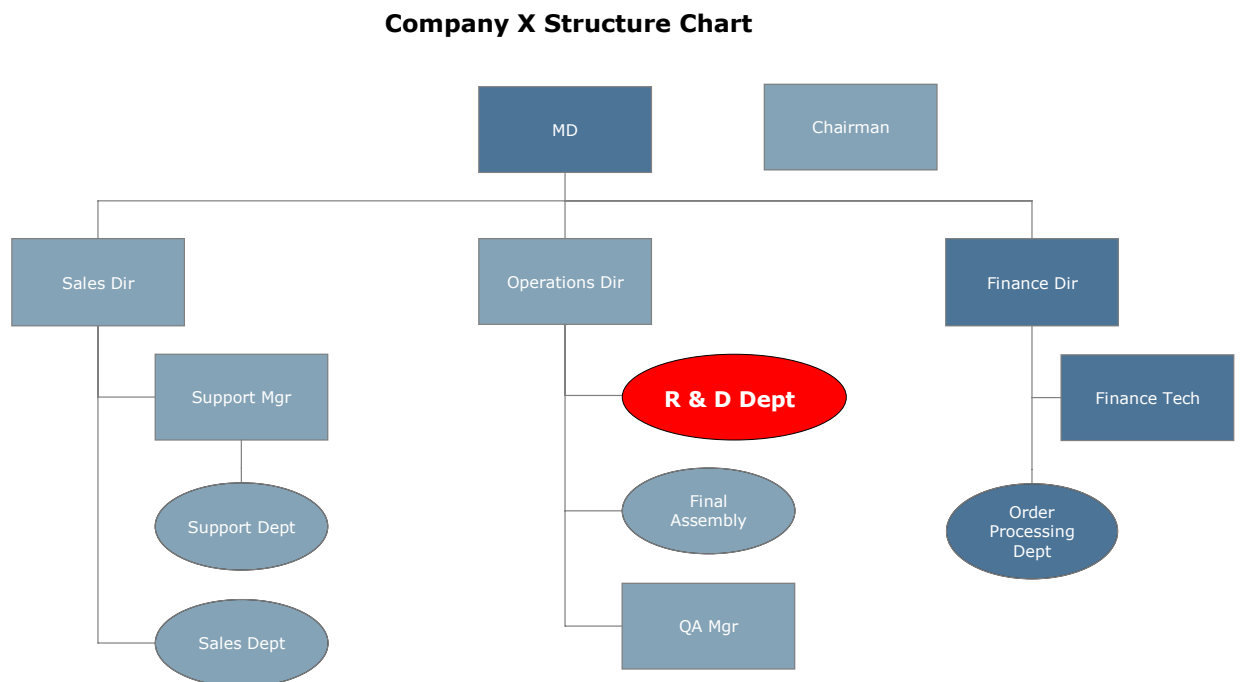


Fig 5.1 Company X Org chart

5.1.1 Industry background

This Irish small to medium enterprise is in the building control industry, providing building management systems (BMS) software and hardware solutions. The company will be referred to as company X. Company X produces hardware, firmware and software that controls HVAC – heating, ventilation and air conditioning (HVAC). The target markets that it sells to are predominantly European. It commands 70% of the Irish market and between 2 – 6% for the UK and mainland European countries. The company produces and sells building control hardware solutions and BMS software. The solution comprises of Input / Output electronic controllers and proprietary software. The software interfaces with the controllers and also programs them. The controllers can work standalone and control a building but initially they must be programmed using the PC based software. The controller's consist of printed circuit boards with communications interfaces and electronic input and output sensors and controlling devices. They are housed in plastic and powered by external 24 volts alternating current. The controllers function is based on data collection from the building environment and output calculation to control the environment. The BMS software is installed and executes on a designated PC in the building to be monitored. The software and controllers are typically accessible on the local Ethernet network. See figure 5.2 for a basic BMS system.

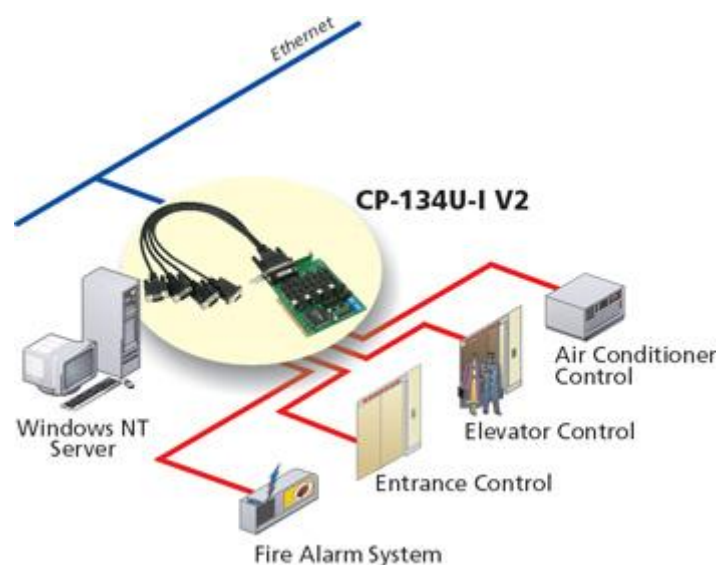


Fig 5.2 Example BMS system

5.1.2 Description of company X BMS system

To further explain the operation of the BMS and controllers, I have extracted the principal components, and offered a description which builds up the working model of a typical BMS system.

There are three levels of the BMS, principally the BMS PC, communications controllers and field controllers. The BMS PC would interface with one designated communications controller. A building may contain many hundreds of communications controllers and a multitude more field controllers (perhaps 10 or 20 times more). The communications controllers are accessible on the Ethernet network of the building. Each communications controller has its own unique address, and share the same network. A number of field controllers (e.g. 15) are in turn controlled by one communications controller. The field controllers operate on their own sub network, each having a unique address. See figure 5.3 for a network of controllers and BMS PC.

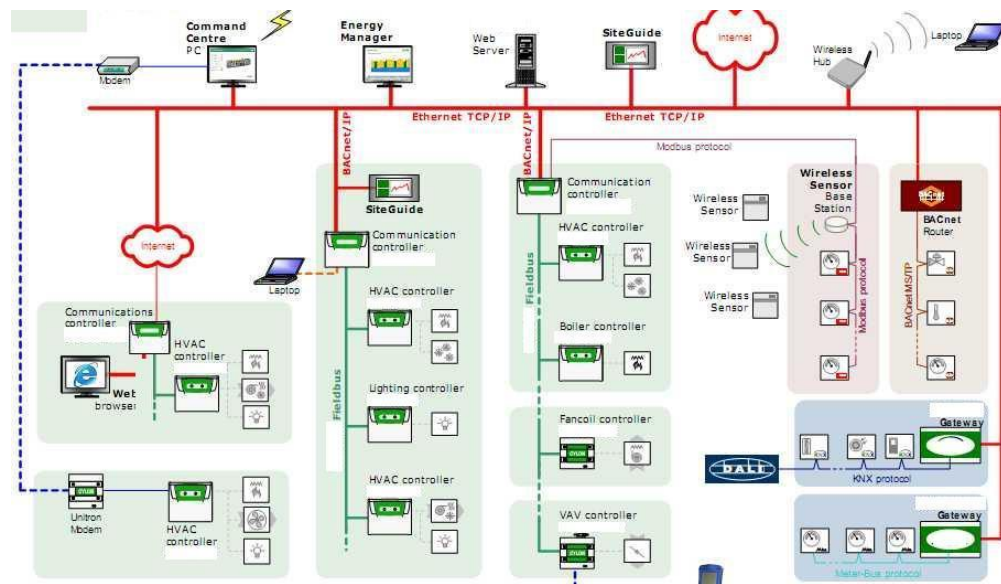


Fig 5.3 High level architecture diagram of accompany X's solution

The field controllers are those controllers which directly control the building environment or system (e.g. Lighting, Heating). The field controllers receive data in both analogue and digital formats. The data is received directly from input signals

from sensors in the building environment or indirectly from another field controller. The controller calculates an output operation based on these inputs. The output operation is determined by the controller's strategy, the strategy is a program executing on the controller itself. This output was used to regulate the building environment. The output signal is sent to actuators which operate the plant machinery which in turn regulate the building environment e.g. heating.

Each field controller must be programmed directly for each building environment; this program is referred to as an engineering strategy. This programming is achieved via a PC based computer aided software engineering (CASE) tool. This engineering strategy must be downloaded to each controller via the PC and controller communications network. The field controller's strategy is devised by an engineer specialising in the HVAC industry. The strategy is fundamentally a program of mathematical calculations that the controller executes. In the example of a heating system the calculations are based on the input sensors plugged into the controller (e.g. heat sensors) and output actuators (ignition circuits for gas boilers, water pumps, fan coils). The goal is to control the heating system based on the temperature of a room, and the necessity of heating or cooling the room via a fan coil or other heat device. The strategy creation is graphical for the engineer on the engineering application and BMS PC. See figure 5.4 for a small strategy example.

The Engineering strategy has 5 principle components:

1. It is a graphical representation of input and output points connected to a mathematical module.
2. The points are unique for each field controller.
3. Points may be virtual and broadcast from one field controller to another via a communications controller.
4. The modules and points can be edited and saved multiple times.
5. The strategy is saved in a format that can be downloaded and executed on the controllers.

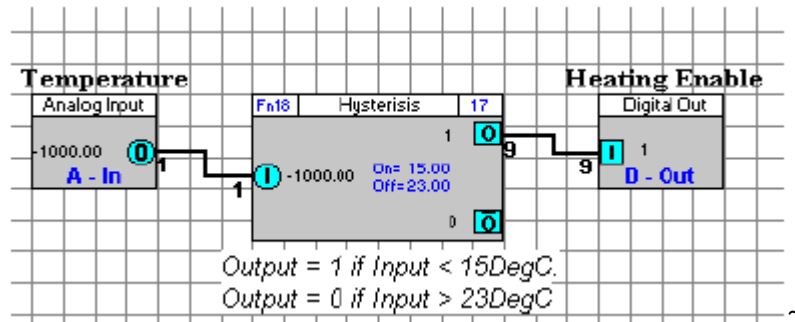


Fig 5.4 engineering strategy example

When the strategy has been completed and downloaded to each field controller, the controllers are monitored in situ via a suite of applications on the buildings Ethernet network. An interfacing communications application typically would control communications between the controller's proprietary protocol and that of the IT network e.g. using Ethernet or other network protocol (e.g. BACnet or Modbus).

The controller hardware, firmware and all other supporting software are designed and written in the company's R&D department. It is the responsibility of the QA department to test the firmware and all software before release to customers. Technical support offers training and support to the customers. The customers are represented by two sectors of the industry. The end users are those that ultimately monitor the buildings and plant installations. The installers are those customers who purchase from the company and act as intermediaries and install the system solution for the end user.

5.1.3 Research and Development department description

There are four teams in the R&D department. Each reports to the Operations Director. The Hardware team design the controller circuit boards and input and output interfacing devices. The Firmware team design and code the firmware for each controller. The Software team are responsible for testing and the QA of all releases from the R&D department. See figure 5.5 for an organisation chart for the R&D department. My role is within the test team as its leader. I am directly responsible for all releases from R&D.

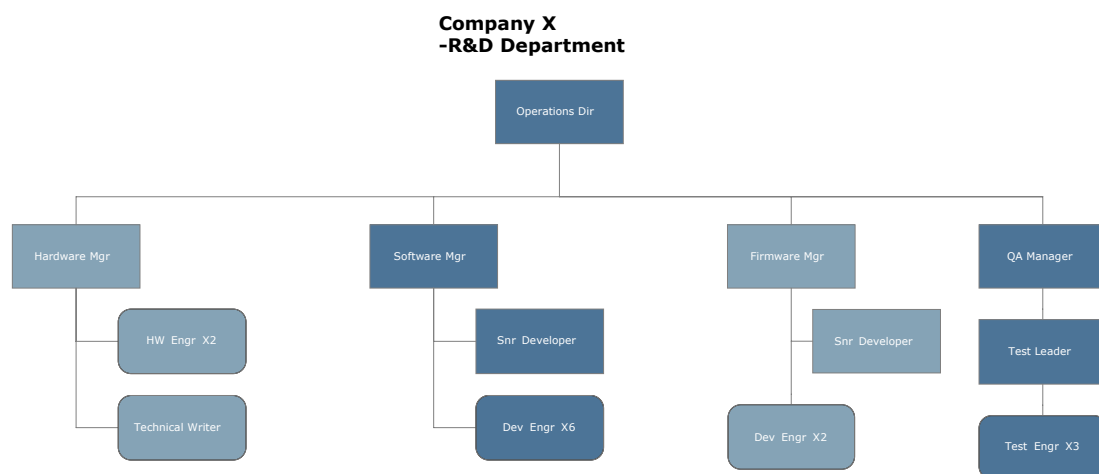


Fig 5.5 Company X R&D Department Org

The QA department have two software releases and one firmware release per annum. The software consists of three principal and twelve ancillary applications. The principal applications consist of a database server, a communications interface application (Port handler) and an engineering application. The port handler allows the controllers to communicate with the software in real-time. The engineering application facilitates the controllers to be programmed for each target building plant and associated sensors which the controllers control. The ancillary applications assist with the monitoring of the controllers. The releases consist of those applications which are modified in the form of maintenance / enhancements with bug fixes and are bundled with a windows installer on a CD. The firmware is shipped embedded in ROM chips for replacement. The QA department are responsible for testing and certain QA functions for R&D, it is best to group both functions into this one department.

The diagram below outlines at a low level the architecture of the software suite for company X. The diagram consists of a small controller network and the structure of the main applications which comprise the BMS suite.

In the centre is a communications bus, this DDE communications bus facilitates application communication with each other and also the controllers via the port-handler. The port-handler communicates directly with the communications controllers, the other applications communicate via DDE with the port-handler. The flat files and databases associated with each application are also pictured.

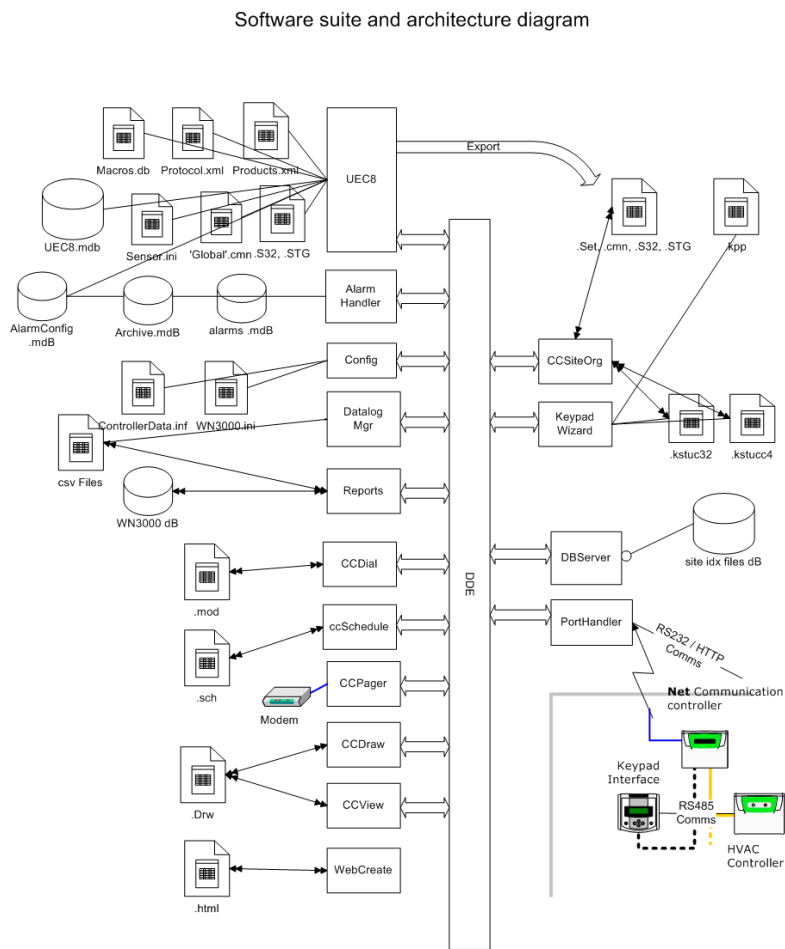


Fig 5.6 Low level software suite and architecture diagram

The building control industry has in the past been slower to evolve than other areas of commerce. However this has changed dramatically in the last few years. The pace at which information technology has spread to every area of commerce has resulted in broad industry requirements to keep apace of this ever-changing sector. The HVAC industry has also been quick to embrace the benefits that the latest IT solutions can provide. As a result there has been quite a heavy demand for innovative software and more efficient yet complex systems for this industry.

To meet these industry requirements the company undertook the development of a new Engineering Tool application and two new controller types to replace all existing controllers and for the development of supporting firmware and software. The expected lifespan of these new products was expected to be ten to fifteen years.

At the time of conducting this research, a new engineering application was released to customers. This was the first new product of a scheduled three major releases. It did not meet with customers expectations. The next product to be released was falling behind schedule. As a result of these problems it was necessary to conduct research into best industry test and QA practices with the intention of making changes to the Quality Assurance effort of R&D.

5.2 The Quality problem that is to be tackled

At the time of this thesis there were two main issues in the company. The immediate concern was in respect of the quality of released software; the engineering tool application and ancillary applications. The secondary concern was the rate of progress of the first of the two new controllers in development.

The engineering application was the fundamental software required for the programming and operation of the controllers. The controllers and indeed the company were dependent on this application for operation and success. Following from its release customers had reported a worrying number of failures of the software.

The company had a quality system in place and this was certified to ISO9000 standard, “ISO 9000-3, the guidelines offered by the International Organisations for Standardisation (ISO), represent implementation of the general methodology of Quality Management ISO 9000 Standards to the special case of software development and maintenance” (Daniel Galin, 2004. p 477).The company was also audited annually by an external consultancy firm to retain its certification. This certification related to the company quality procedures and their execution and not to the quality of its products. My responsibility was to ensure that a quality product was released to all customers. I found that the ISO Quality System was failing in this regard. In my opinion, action was required and a better understanding of the problem was necessary. An analysis of the problems lay with working with the customer’s issues and tracking their cause backwards from release back through the quality system to the project inception.

5.2.1 The investigation

A proposal for improvements was made to the company directors in relation to testing practice improvements and on quality process changes. The proposal was based on an assessment of the company's software quality process and on the engineering application project. The assessment was conducted in a similar format to the principles of the SEI and CMM processes assessment and also in line with the company internal audit process. The assessment was conducted to investigate the nature of the quality problems and what process improvements were required. The company documents listing the assessment and changes can be found in Appendix A.

The assessment was conducted in three phases:

Planning of assessment	Identify what departments / teams are to be assessed. Assess the participants of each department. Solicit customer feedback for the engineering application from the marketing department. Prepare for the review of department process documentation. Prepare for the review of department project documentation. Prepare for an interview with each department manager. Schedule a time for the assessments for each department.
Assessment of each department	Assess each participant manager before assessing their department and interviewing them. Review process documentation. Review project documentation. Conduct the interview of team members and managers . Document the findings of each team assessment. Agree with each team manager the findings and ratings of the assessment and obtain sign off from team managers.
Reporting of findings	Compile and present the findings of the assessment. Act on the findings and plan quality process improvements.

5.2.2 The investigation findings

The assessment was conducted following the three stages outlined earlier. The initial assessment findings are grouped into five distinct areas:

1. The overall defect statistics for the released engineering application project from all departments
2. A quality report from customer support based on customer feedback on the released engineering application project
3. Test case design and test planning for the engineering application project
4. An internal audit of both software, test, support and firmware departments in terms of the engineering application project and processes
5. An assessment of the development life-cycle and quality system process in general following from the previous 4 assessment areas.

Each of the five areas is discussed in detail over the next few pages followed with the proposed solution.

1. Defect statistics for the engineering application

The engineering application took approximately 960 man days of development effort and 360 man days of testing. It was two years late to market and its reception by customers was not positive. This application had approximately 61,254 Lines of code and approximately 59 Function Points per KLOC (FP/KLOC) which is above the median (53) for a Visual C++ application (Quantitative Software Management, 2005, internet). Defect analysis revealed that it had a defect rate of 1.5 per Function Point. This is twice the defect rate for a CMM level 1 company (0.75) and thirty times that of a CMM level 5 company's (0.05). The function point analysis matrix (see chapter 4 for more details) and defect breakdown are listed overleaf.

Σw	$FC = \Sigma w * x$	General Characteristics @	Weight c (0-5)	$VAF = 0.65 + .01 \Sigma c$	$FP = FC * VAF$
12	168	1. Data Communications	5	0.7	113
18	252	2. Distributed functions	1	0.66	171
50	700	3. Performance	1	0.66	472
35	490	4. Heavily used configurations	2	0.67	330
8	112	5. Transaction rate	2	0.67	75
0	0	6. Online data entry	3	0.68	0
0	0	7. End user efficiency	2	0.67	0
0	0	8. Online update	2	0.67	0
0	0	9. Complex processing	2	0.67	0
0	0	10. Re-usability	2	0.67	0
0	0	11. Installation ease	1	0.66	0
0	0	12. Operational ease	4	0.69	0
0	0	13. Multiple sites	3	0.68	0
0	0	14. Facilitation of change	4	0.69	0
123	1722	0	34	0.674285714	1161

KLOC	61
Number of defects	1772
Number of FP	1161
FP per KLOC	19
Defects / FP	1.526112719
Average LOC/FP	52.53548298
Defects /KLOC	29.04918033

Fig 5.8 defect metrics for engineering application in terms of FP and KLOC

Fig 5.7 function point analysis findings

After 18 months of in house testing, there were 1772 defects recorded, only 55% had a recorded severity rating. There were over 164 defects found during the Beta testing of the application by customers. There were another 62 latent defects found by customers after the full release. The in house defects yield a ratio of 1.46 defects per function point or 29 defects per KLOC.

The new HVAC controller in development was already 6 months behind schedule and early testing results were not positive. The firmware for these controllers was a migration of existing code to a new embedded chip. The HVAC controller had approximately 23 FP/LOC. Since this was a migration of existing code it was not evaluated in as much detail.

2. Customer Questionnaire Feedback

Twenty one customers were contacted for their feedback on their experiences of the Engineering application. Out of those customers contacted eleven or 67% responded with serious problems during usage. 15% of customers refused to continue use the application based on their bad experiences. 67% of customers who had serious defects in the first six months of use reported a total of 62 defects. There were five common functions of the application all the respondents made reference to. The main areas are listed below:

- a) The set up and use of broadcast points across the controllers communications network
- b) The use of macros for reusing engineering strategies
- c) The use of virtual points to reuse existing points in the engineering strategy
- d) The use of Printer Scaling to print out a copy of the strategy
- e) The occurrence of duplicate points in the engineering strategy

a) **Broadcast points**

The purpose of the broadcast points in the engineering strategy is to facilitate multiple uses of a single point across the building environmental control. There may be several hundred Net controllers with each one having upwards of 64 HVAC controllers. A single point may be broadcast to hundreds of controllers on the network.

The setup and use of broadcast points from the engineering strategy was working correctly but the editing of existing points was causing the corruption of existing points. Existing points were being over written in the '.cmn' file. When this file was downloaded to the controllers, the building environment could not be controlled properly. The result of this was that the strategy and building environment were in an unstable state, it was also very costly for engineers to troubleshoot and to rectify this problem.

b) Use of Macros

The purpose of Macros in the engineering centre is to facilitate the reuse of more common strategy elements. A macro can be thought of as a copy of a small strategy. The problem with the macros in the engineering centre lay with the number of combinations of sub components which could be created; there were certain combinations of components and the order in which they were used which caused the macro to become corrupt and unworkable. The problem was not noticeable during the macro creation or edition; it only became apparent after macros were used in the strategies at a later stage. The problem left strategies in a corrupt state and unusable when downloaded to the controllers. Once again this was a costly problem to rectify for engineers.

c) Virtual Points

Virtual points were created to allow engineers have the flexibility of using additional points during the creation of the engineering strategy. Virtual points themselves in the strategy meant that there may not necessarily be actual physical inputs or output points on the controller itself, they would later be broadcast from another controller. The benefit of virtual points in a strategy allowed a one to many and a many to one connection between strategy modules.

The problem with the virtual points was that when they were edited or used in a certain way the one to many combinations of the point numbers were changed from their original state to that of a new state. The problem for the engineers was that extra care was needed with their use to prevent the change of existing strategy work, and for them to come up with a new way to allow for their strategies to work as they expected. Once again it was a costly and time consuming exercise for the engineers.

d) Printer Scaling

The engineering strategy was created on a graphical CASE tool and the strategy was stored in a format that was communicated to the controllers but also visible in a graphical environment. Printer Scaling allowed the completed engineering strategy to be resized to a visible level in the graphical environment that was both legible to users on screen and printable on single sheets of paper. The printer scaling function didn't work effectively (it was in a MS bmp format) to allow both of these requirements. The result of this oversight left engineers being unable to print out the strategies for their customers records. The printed strategy was a record of completed work and then used for calculating maintenance work. The printout served as a blueprint of the completed HVAC system. If it was not available or legible the engineering firm who commissioned the building were then liable for additional maintenance effort and cost.

e) Duplicate points

One of the most critical problems with the engineering tool was the occurrence of duplicate points appearing in the strategy. During the course of editing a strategy under certain circumstances, the occurrence of duplicate points arose. The changes to the original strategy files were downloaded to the controllers via the port handler and caused the malfunction of the building management HVAC system. These duplicate points were not obvious during the graphical editing of the strategy; they were not also obvious once the strategy was downloaded to the controllers. It was only when a problem manifested itself and investigations were underway that they were detected. The principle reason behind the existence of duplicate points arose when an existing point number was changed to another number, or the point removed and reinserted. The point would take up a number that was already allocated on the strategy and thus corrupt the strategy. This corrupt strategy was then downloaded, the consequences were that the HVAC system was not in a stable state, the time and cost to identify and rectify the number of strategies was considerable.

3. Test case design and test planning

Following on from analysing the defect statistics and customers questionnaire responses for the Engineering application, the test cases and test planning were placed under scrutiny. The major areas of concern were evaluated with respect to the test cases to ascertain if the tests would have been adequate to detect the defects in those areas. The test planning was also examined to see if the plans took into consideration a systematic approach to the testing of the application.

The test cases for the released Engineering application were found to be inadequate. Out of 128 pages of tests, 32 pages were functional tests to validate the correct operation of the user interface. There were no boundary tests or explorative testing of functionality. The remaining tests were regression tests for the defects that were detected during the course of testing to ensure that all defects were fixed.

In terms of test coverage of the functionality of the engineering application the existing test cases did not cover any white or grey box testing. There were no higher order tests such as performance tests or usability tests.

In relation to test planning there was no consideration of integration testing with other elements of the ancillary applications or the firmware communications. There were no system tests. There was no test data or an environment that matched that of customers in which to execute the tests. The result of this analysis was that the tests were not adequate in terms of how the application was structured and how it would be used by engineers on a daily basis. The existing test plan meant that the average time taken to execute these test cases was approximately 40 man days of testing for a complete test of a build with the existing test cases. The test environment and test data were also not reusable and added to the overhead of each test cycle.

In terms of quality, there were a total of 81 builds and an unknown number of test cycles executed before the release of the application to Beta customers. The quality of the application before it was released was not known in any degree of certainty.

4. Internal audit departmental findings

Three of the R&D department teams and one of the customer teams were audited by an ISO and Tick-IT certified internal auditor. The audit was carried out in accordance with the company ISO 9000-3 guidelines for internal audits and as part of the investigation for the proposal for QA improvement. The teams audited were software, firmware, test and customer support. These teams were audited specifically on three topics:

1. For the team compliance with existing company quality standards in terms of the Engineering application project documentation.
2. On the effectiveness of the existing company quality procedures for successful project execution for this team
3. For suggested improvements for improved quality from other team to this team

An internal auditor (the author of this thesis) audited each of the departments to identify problems; the audit was followed in accordance with ISO guidelines. During the audit each team manager was interviewed so that the project documentation non conformances could be discussed and agreed. During the interview the effectiveness of the existing quality procedures for the respective team was discussed with each manager. The managers were also asked for suggested improvements in terms of project quality of the project deliverables that they received from other teams and on improvements that they could make as team deliverables. The findings of the audits regarding compliance with existing company quality standards for each team are listed overleaf.

Software	<p>Low priority defects were not fixed</p> <p>There were no design documents for some of the engineering applications functionality.</p> <p>There were no customer requirements for some the engineering applications functionality.</p>
Firmware	<p>Specification documents were not kept up to date resulting in a number of latent defects.</p> <p>A number of customer requirements were not implemented in the firmware for the HVAC controller project.</p> <p>The project schedules were not tracked or updated with project progress.</p>
Test	<p>The test procedures were unsatisfactory for a real-time embedded systems software company since the testing failed to detect numerous serious defects in the engineering application</p>
Customer Support	<p>There was no beta test plan put in place for the testing of the engineering application.</p> <p>There was no record kept of beta customer's details and defects.</p>

5. Assessment of the development lifecycle and quality system

An assessment of the overall procedures of the software, firmware, customer support and test departments was conducted from the perspective of an interdepartmental and enveloping software development process. The findings for this assessment of the development process were as follows:

1. The current waterfall development life-cycle that is implemented fails to include the test department and customer support until after the code has been complete.
2. The test process is inadequate to test the functionality of the software and firmware at a detailed functional level and structural perspective. The test process does not allow sufficient time or resources for system testing or testing that is representative of customer's expectations of a quality product.
3. There is no review or sign off of requirements or design documents by departments other than the software and firmware departments.

The assessment was conducted to identify the root causes of the problems that the company was facing with the engineering application and current projects in development. The original problems with the software and their root causes are outlined below.

- a) Late release of the software
- b) Software not meeting customer's requirements
- c) Latent defects still present in the software
- d) Technical support and customers are not aware of all new features present

a) Late release of the software

- New features are requested from customers during the development of the software. These change requests require rework and are not rescheduled. This activity puts increased pressure on the developers to meet the deadline and the consequence is a late release.
- Priorities change on other projects and a developer or tester may be required to work on a different project.
- There is no formal handover of products for release to technical support. They do not start their acceptance testing at the time of handover.
- There is insufficient time allowed for testing. The estimated time for testing in the schedule is inaccurate.
- The test cases were inadequately designed for detecting low level defects. No analysis was conducted on the components of the applications or how they interacted. There was no testing of the internal logic of the application.
- The test planning was inadequate to prepare sufficient inputs to the test process. There was no planning for a realistic test environment or for the creation of test data.
- The defect tracking didn't capture the history of any defect changes which would give information for future test planning and defect root cause analysis.
- The lack of version control was also a factor contributing to delaying the release of products.

b) Software not meeting customer's requirements

- The customer's requirements are not fulfilled because they are not captured accurately in the user requirements documentation.
- Another reason for the customer's requirements not being met is that some features are not being implemented, because the team discovered that the work required is more substantial than originally estimated.
- There is no research conducted into likely customers needs, when feature requests are made there is no scope for the seamless accommodation of the requests. The software is modified in any way possible to meet new requirements without consideration for possible side effects.

c) Latent defects still present in the software

- There are bugs still present in the software after a release either because it was decided to release the software and fix the bugs at a later date or because they were not detected in time during testing.
- There was inadequate time and resources planned for the full testing of the software.
- The test cases were not thorough in testing certain components of the software. This was not evident until after the release of the software with customers on live sites.
- The testing was not thorough enough in capturing how a user uses the application. Hence the user detects bugs that the tester overlooked.
- There is insufficient system testing done.

d) Technical support and customers are not aware of all new features present.

- The handover to technical support is not scheduled; as a result the technical support team does not have enough time to prepare support or training for the new software.
- The customer may not have received enough training. The customer may not read the manual or may not consult the help file.

While these problems are specific to one company, they are prevalent in the software industry where similar problems affect the global software industry. According to Robert N. Charette in 'Why software fails' in an article in IEEE the main reasons for software project failures are: (Robert N.Charette 2005, internet)

- Unrealistic or unarticulated project goals
- Inaccurate estimates of needed resources
- Badly defined system requirements
- Poor reporting of the project's status
- Unmanaged risks
- Poor communication among customers, developers, and users
- Inability to handle the project's complexity
- Sloppy development practices
- Poor project management
- Stakeholder politics
- Commercial pressures

5.2.3 The proposal to the company

The main issues from the assessment were reported to the directors of the company with recommendations for changes to processes and practices. A report was compiled which outlined an approach as to how the testing process should be addressed for improved testing in future projects (SW Test Department requirements.doc, Appendix A). The suggested improvements were reported in the form of an official company project documented Engineering Change Request (ECR) for company X (ECR100, Appendix A). The test report and ECR were submitted for approval to the company directors, following the director's approval the changes were scheduled for implementation.

5.3 My proposed solution

Addressing the immediate problems

Following on from the investigation a number of areas in terms of quality needed to be addressed. These included the engineering application defect metrics, addressing the customer's feedback, improving the testing process and the development lifecycle.

“A quality improvement programme leading to the establishment of a quality system must have both technical and cultural aspects, each being equally important. It is easy to see the reason for this: the entity to be improved consists of both technology and people” (Joc Sanders et al. 1994, p19)

5.3.1 The Principal design factors behind my proposed solution

- i. Designing a quality focused project team with the sharing of knowledge and evaluation of each members work with inspections and peer reviews
- ii. Developing and reusing template documents and checklists where possible to improve technical and customer knowledge artefacts
- iii. Assessing Quality from the start and then at each stage of the project with effective defect removal and planned systematic testing.
- iv. Continuous Improvement where possible

Effective use of Team and project knowledge

It is a more efficient use of resources on any project when the experience of an individual is used in a collaborative manner. The ideal is to build up a collaborative team from the start of the project so that the identities and responsibilities of all stakeholders from each department or domain are known at the outset. This will increase the internal communications among the team with a lesser requirement for a manager or mediator.

Where there are differing perspectives and priorities from each team domain at

different stages of development, the project goal is success throughout each stage. Having attendance by each domain representative at meetings should allow for a smooth transition from stage to stage.

By having team participants review and inspect the work of their domain colleagues they learn more about the project from other perspectives while also contributing their experience to the project and improving the quality by removing defects and by reducing defect injection.

A collaborative team assessment at each project stage allows for the input of different perspectives on the same subject matter. When the responsibilities of each team member have been defined early on and each member knows that they must sign off on particular article of work outside of their domain they will then review it with due care and attention. The documents evolve and improve over time with successive inputs from project participants. Ultimately the documents will become more company focused and of a very high standard.

Developing and reusing formal documents and checklists

The development of 'best in class' documents by all domains will assist with ensuring that each domain can understand what is being developed by the other domains and can contribute to the dissemination of project information at each stage. The use of checklists will assist with the review of each development stage and ensure that nothing gets missed in reviews. The documents can be appended to, used for reference by new team members and then reused for other projects.

By developing best in class documents they will over time become templates for later projects and also act as a motivator for participants. The standard in documentation will improve over successive revisions and increase the standard for all team members. Having each domain formally sign off on a document, it increases the attention paid to the documents content and also focuses the team on their respective roles.

The reuse of documents will reduce the project timeframe by having templates already in place. It also acts as a source of information for future project revisions and training material. The documents when used with a process model can act as an interface to external organisations or teams when broader projects are embarked on.

Assessing Quality at each stage

The method of quality assessment should include the review of a team member's work by their peers before its external review and/or inspection by other members of the project team or external experts. This assessment includes the verification of design documentation and the validation of software builds followed by a qualification for the builds release. The record of metrics at each stage places a value on the quality of the project outputs.

Having independent verification and validation at each stage of the project increases the defect detection rate and reduces the defect injection rate. Integrating an independent and quality conscientious team adds an emphasis to quality assessment in each domain. An independent test team which is provided with sufficient application design and business knowledge can plan detailed testing. This knowledge can be used in the design of structured, methodical and reusable test cases for the detection of defects in each stage of the software lifecycle. To guarantee their independence a reporting structure which allows the team to escalate issues outside of the project development team is required.

Continuous improvement

The notion of formal sign off, reviews and assessment may seem to impede the creativity of the individual team members. However creative individuals will always prevail and the framework is open to interpretation. The human element on a project will inevitably lead to mistakes. The collection of metrics in repositories and the generation of reports will allow individuals to learn from their mistakes and for continuous improvement in projects and software development. Failure to record errors will allow them to be forgotten and repeated.

5.3.2 An Initial model

In order to improve the software in the areas of testing and software quality, it was necessary to develop a model that could be used effectively to verify and validate the software at each stage of its development by all parties involved with the project. The first concern was to address the present problems and then to develop the solution model further, by evolving and maturing it over successive projects. The long term solution was directed at producing a framework that could be used repeatedly both in house and in other software company's. This initial model was directed at addressing the immediate concerns of each department in the company

Firmware and Software Development	<p>Write a user requirement document (URD).</p> <p>Document both a high level design (system specification document) and a low level design (technical specification) solution.</p> <p>Document and execute Unit tests.</p> <p>Maintain proper version control of builds and code.</p>
Test	<p>Implement a quality policy for the team</p> <p>Implement a checklist for the review of each document from development to QA.</p> <p>Participate at all documentation reviews for the early stages of the project and log issues in a repository raised during the reviews</p> <p>Produce a test strategy and plan for the project and seek peer and project approval for each test artefact.</p> <p>Insist on effective configuration management of the builds that are tested and released to QA.</p> <p>Manage and report the tests executed and defects detected during test execution to the project manager.</p> <p>Conduct defect triage meetings to prioritise defect fixing with development</p>
Customer Support	<p>Review the user requirements document and proposed design solutions at an early stage of the life-cycle.</p> <p>Collaborate with the development lead during use cases creation.</p> <p>Document acceptance tests and have these reviewed by QA before</p>

	<p>accepting a release.</p> <p>Report accurately all defects during beta or acceptance testing of beta software.</p> <p>Report accurately all defects detected by customers on released software.</p>
--	---

Testing improvements

To address the immediate problems with testing, a test process was developed which included template documents for the effective planning of testing. The test process documents were designed in addition to new process documents from the development departments. Each document was to be peer reviewed by the author's colleagues before being subjected to an interdepartmental review. Following on from the review, any open topics raised at the review would be followed up by the author as action points. The author would follow up on those action points with a second review or send out an updated document with amendments as appropriate.

The initial documents from development would be a user requirements document (URD) and system specifications document (SDS). The URD would record what was required from a software solution and the SDS how that solution was going to be implemented.

The information contained in the URD, SDS would be used to create the test strategy document. The test strategy documents purpose would be to describe at a high level what the test approach to the project will be. The test strategy would also be used as a matrix to map user requirements and design specification points to tests. The test strategy would also allow the test lead to make preparations for a test environment and to source requirements for adequate test data.

The test plan would be a more detailed explanation of the testing approach and act as a test schedule for the project. It would expand on each of the areas of the test strategy but also set out in more detail each of the points from the URD and SDS in terms of what will be tested and when. The test data and test environment details would also be documented in advance of their configuration in the test plan.

The test execution process would follow the test planning process. The first part of this process would be a formal handover of builds from development to test. This handover would maintain the requirement for version control of both code and builds. The details of defect fixes would form part of this handover form. This would assist with the defect management and regression testing and maintain the test status of different builds. Bug fix reports were to be completed and compiled together and form part of the handover of builds to QA. This configuration management practice assisted with the quality assurance of individual builds. The test results would be documented with defect states so that progress reports could be compiled with an assessment of software quality. Metrics for test case completion and defects per component and build would be recorded to assist with the identification of the root cause of defects. This information would be factored in for quality improvement in successive projects.

Following QA test execution completion and signoff, the application was handed to the customer support department to conduct user acceptance tests either on site with approved Beta customers and or in house.

The processes were documented and included in the company's' quality system. The company documentation for this process can be found in appendix A. A diagram of the test process and test execution procedures is depicted below in figure 5.9.

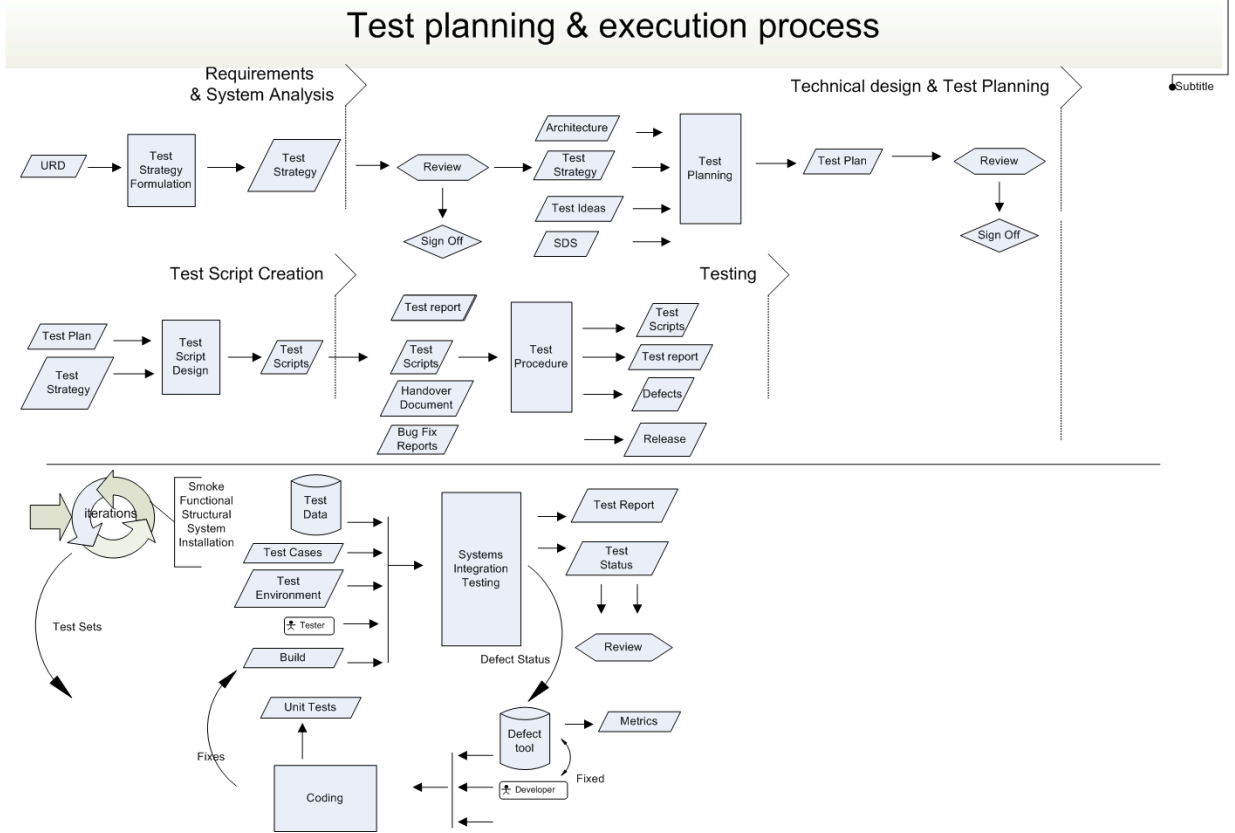


Fig 5.9 improved test and QA process

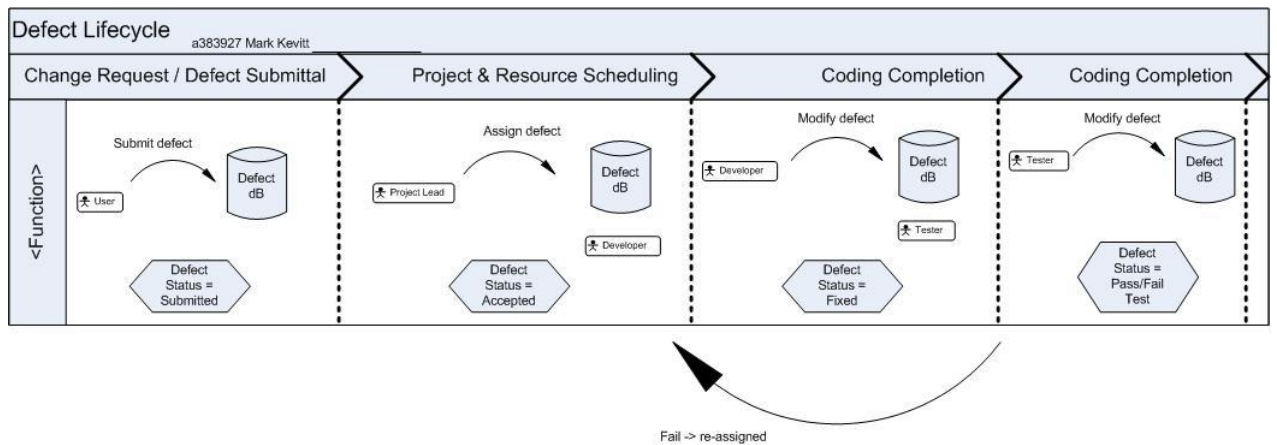


Fig 5.10 Defect Lifecycle

The company's quality system was changed to reflect the new test process and supporting development documentation. The documents that were changed to the company quality system are included in the appendix A. The documents are listed below:

ECR – 0100 Testing Research Plan

Procedure 0029 writing test documents

Procedure 0056 software testing procedure

Work instruction 0032 test script creation

Work instruction 0005 dealing with an incident in released software

Work instruction 0081 use of Bugzilla for defect tracking

Form 0105 software handover form

Form 0123 firmware handover form

Form 0127 SW test report form

5.4 Summary

In this chapter the industry sector that the company operates in was explored. Elements of the company, specifically the R&D department, were described in full. The function of the R&D teams and the company products were described in terms of the BMS architecture. The architecture and operation was explained in depth. In relation to the BMS system, the fundamental of the engineering application was explained and its role in the BMS system put into context.

The fact that the company was experiencing quality problems was mentioned. Investigations into the quality problem were conducted in an assessment. The assessment findings were explained in detail. Details of the findings included feedback from customers, internal departmental and company quality process audits.

The root causes of the assessment findings was compiled into an engineering change request / report which outlined a proposed solution to the problem. The proposed solution was described and depicted in a graphical process. The process included testing and quality assurance practice improvements. These improvements were to be implemented and evaluated over forthcoming company projects. This implementation is described in the next chapter.

6 Chapter Six - Implementation of improvements

In chapter five, I outlined the quality problem in the company; I also described the proposed solution to improve the quality problem. In this chapter the proposed solution is implemented and evaluated over three successive projects, each project is executed in succession. The test & QA practices during each project were further improved following each project as per the action research spiral (See chapter 2 for details). The projects were executed over a three year time frame. To recap the seven phases in the action research cycle 'Action Research in the Organisation' are:

1. Review the current practice
2. Identify an aspect that needs to be improved
3. Plan an improved practice
4. Act / Execution of the practice over the course of the project
5. Observe the effects of the practice
6. Reflect on the success or failure of the practice and re plan accordingly
7. Repeat the practice improvements until complete

The first three phases were performed prior to the proposed solution implementation. They are described in chapter 5, this chapter deals with the remaining phases (4 – 7). Each project is described under the following headings:

- Description
- Plan
- Implementation (Execution and Observation)
- Reflection

The first three projects were conducted for company X with different development teams on projects of similar size and complexity. There were approximately 500 function points per firmware product and 1500 function points per engineering application and the number of lines of code was 22K for firmware and between 50K and 70K for the different engineering application versions.

The table below gives an indication as to the different size and complexities of the three different projects for company X.

	Project	Project	Project
	1	2	3
FP	~1200	~600	~1600
KLOC	60K	22K	225K

Fig 6.1 project size and complexities

The duration of the projects differed with varying numbers of resources on each project with different skill levels. Since the projects were tested at various stages of development and were roughly equal in size the number of defects are not divided by the KLOC as this was indeterminate at the stage of testing.

6.1 Company X - HVAC Controller Project

6.1.1 HVAC Project description

To address the immediate problems facing the software in the HVAC Company, the released engineering tool application code which was most problematic as identified by the customers had to be re-developed or 're-factored' to address the outstanding defects. The HVAC firmware was also behind schedule and needed to be completed. It was decided that the next release of software had to be a defect-free engineering application, the new HVAC firmware and all of the ancillary applications modified to support the new HVAC firmware. This effort was to be included in the HVAC project with an eighteen month timeframe.

In all, there were 11 ancillary applications to be enhanced with one major application rewrite, one new application and the completion of the HVAC firmware. There were eight developers and four QA resources assigned to the project. This consisted of a total of 4500 man day's work. There was approximately 3KLOC to be developed for each ancillary application to allow for them to be used with the HVAC controller. This equated to close on 24KLOC. This was maintenance development which was more time consuming and expensive than new developments. The engineering application which needed re-factoring would require its existing 61KLOC to be re-factored with an additional 8KLOC for HVAC support. The new keypad application would require 162FP and 19KLOC. The controlling firmware were developed in C and the windows applications in Visual C++.

Project 1	
Developers	8
Testers	4
Man Hours	4500
Ancillary apps	24KLOC
Eng App	8KLOC
Keypad	19KLOC

Fig 6.2 Project 1 resources

The HVAC project was the first project to be subjected to the new testing practices. Initially the testing of the engineering application and HVAC firmware was to be the first partial project to undergo the new practices.

6.1.2 HVAC Plan

The purpose of the initial changes was twofold, primarily to test the applications more effectively and secondly to determine the effectiveness of the improved testing practices.

The main changes to the testing practices included

1. Detailed test planning to identify the application components and then determining the test types to be executed for each component.
2. A risk based approach was taken to the priority of the functionality for customers and the complexity of the code that was being added.
3. A thread testing approach was taken where testing would be scheduled for the earlier modules that completed development. This was synchronised with firmware and software so that both could be tested close to the same time.
4. These tests were scheduled with milestone releases from development.
5. The tests were designed for more effective test coverage of the functionality.
6. The tests were also supplemented with detailed test data and an environment that simulated a customer's site.
7. A purposeful defect tracking tool was installed for the recording of defects.
8. The team members were assigned to the project in roles and assigned responsibilities on a par with their experience.
9. Team meetings were planned at milestone intervals to discuss project progress and for the discussion of problems from respective department perspectives.
10. Documents were devised which formalized the interaction between departments and acted as records for project progress.
11. The documentation supporting these new practices was written, circulated and approved before being placed in the company's Quality System.

Test Planning

In accordance with the new process the testing strategy and plan were documented; these consisted of listing the items that required testing and prioritising the applications under test in terms of risk with the higher risk items scheduled to be completed first. The test effort for this project were quite substantial; the test phases identified included the testing of the HVAC firmware, integration, systems testing each application with the new HVAC controller and a regression test of the software with the legacy controllers. The integration testing included serviceability testing each application for correct operation with the HVAC protocol. The system testing included performance testing the HVAC controllers for data throughput and the engineering application for multiple strategy operation.

A project schedule was compiled indicating the roles and responsibilities of the test team. The applications delivery to test and the testing dates were milestones in the schedule. A thread testing approach was taken with a 3 week lag of testing behind development, see chapter 3 the section on 'integration testing'. The reason for this choice was that it was imperative that the high risk items be completed for a release; the ancillary applications could wait for a second release if necessary. The adoption of thread testing meant that some features that were coded could be tested on Alpha builds before the completion of all coding. This over lap of testing and development efforts would allow for defects to be fixed while the developer was still in the middle of coding on the same application. There would be three stages of testing the suite of software.

The phase I testing consisted of integration testing the firmware, communications application (port-handler) and the engineering application. The reason for this initial phase was that the controller had its own proprietary communications protocol. This protocol had to be verified before any other applications could be developed or tested.

The phase II testing was the integration testing of the ancillary applications with the modifications for the HVAC protocol and additional functionality. The third phase of testing would be system testing the entire software suite in a simulated customer environment.

To complete this project the engineering application was the central focus for the new processes and practices. The engineering application needed to be rewritten to address the customer's outstanding concerns. Since the test cases for the existing engineering application were inadequate a new set of tests needed to be designed in addition to detailed test data and a test environment.

A stage progress meeting was held to discuss the current state of the project and what the next phases were going to be. The schedule was reviewed and imposed upon all departments, before the testing was scheduled to begin.

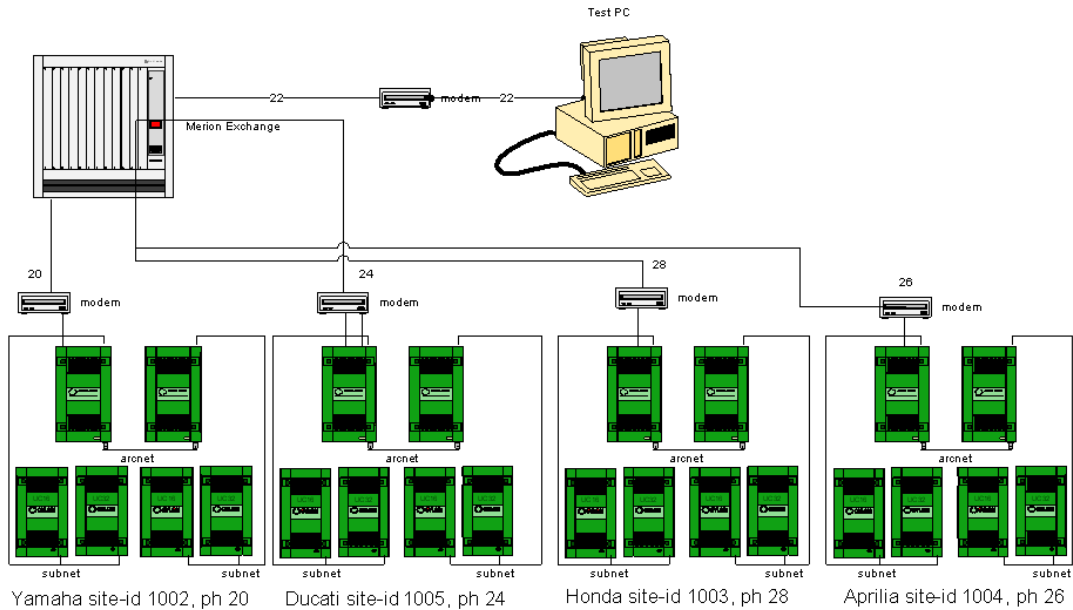
Test Environment

A purpose built test area was necessary to incorporate the new HVAC protocol alongside the existing UC16 protocol. The test area would need to be large enough to have sufficient communications throughput to match that of a large customer's site. The configuration of the test area also had to combine the different protocol communications of both old and new protocols. This required different controller types to be set up in a variety of combinations. Each of the propriety HVAC protocols also had to be tested in each of RS232 Serial and TCP/IP transmission formats to validate the communications functionality in the firmware. Figure 6.3 displays the basic configuration required to test each combination.

The number of controllers and their address ranges were determined to simulate a larger site of full address ranges. The site was configured to include the testing of the communications application protocol and ancillary applications over serial RS232 and TCP/IP transmission protocols.

Software Test Dept - Test Area

Four modem remote sites linked up to the test PC via the merion exchange and 5 modems. The buildings RJ11 network is used to connect the test PC to the merion exchange. Each remote site consists of 2 CNet's connected via arcnet and two HVAC.16's and 2HVAC.32's on each CNet. The Hvac controllers are connected to their controlling Cnet on that CNet's subnet.



Three remote TCP/IP sites are linked up to the ethernet network via the buildings RJ45 network. Each site differs in the number of controllers, but basically one CNet of each site is connected to the Chase Iolan hub. The TCP/IP address of the hub is 192.168.0.30. Each site has a unique port number starting at 10001. Each site is to have as a minimum 2 CNet's connected via Arcnet, 4 Hvac.16's and 4 Hvac.32's. Each pair of Hvac are connected to the CNet subnet.

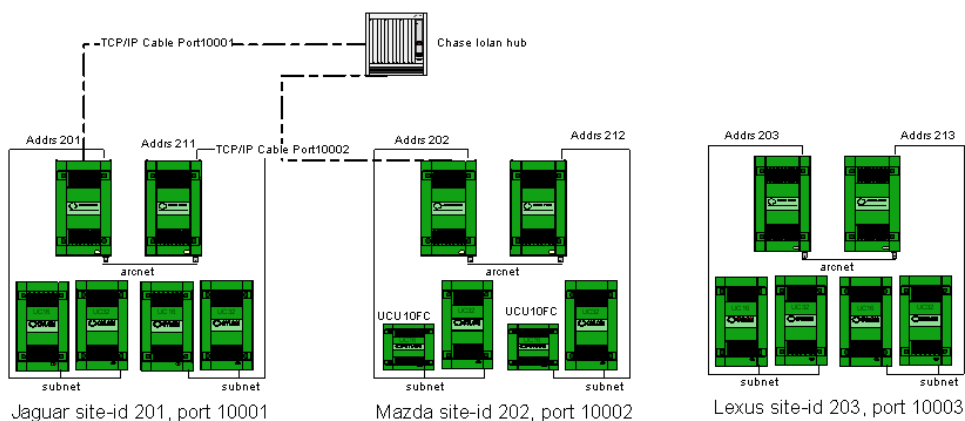


Figure 6.3 Test Area topology

Test Case Design

The previous user interface style of test cases for the engineering application and indeed other applications proved that they were inadequate to test effectively, refer to chapter 5 'root causes of the problems'. Since none of the existing applications had detailed design documents; the existing documentation was in the form of requirements. A new approach to test case design was taken; this approach was based on a combination of black box and grey box testing where the user requirements were supplemented with a system design document of the application provided by the developer. The description of the application was combined with the requirements to list key components of the application. These components were identified as items that could be tested independently of each other as much as possible. The applications were to be component tested to allow for easier regression testing and for better test case maintenance. It was planned that there would be thread testing of the applications with firmware. There was an anticipation of an overlap of certain components being tested while other components were either still in development or defective components were being fixed. The benefits to this component based testing were twofold. Newly completed components could be tested while development of others was still ongoing. Subsequent builds could contain fixes to defective components and also contain newly developed components. This facilitated partial component testing and for testing defect fixes. It was intended that this would improve the efficiency of the development – testing cycles.

The components were identified and listed for inclusion in the test plan. Following on from the test plan the test cases began with all identified components and expanded each component with a number of tests. The tests were designed with boundary value analysis to ensure that the code functioned correctly in likely scenarios and also that it handled unlikely events. Equivalent partitioning was used to reduce the number of tests to a minimum - yet providing for maximum test coverage. A backup of the test data and environment required for the test cases was saved so that the same test conditions could be easily reproduced.

The test cases were designed with efficient defect tracking in mind. If any defects were found they were to be recorded in a defect repository and the defect number

recorded adjacent to the tests. This assisted with retesting the functionality in a subsequent build with the defect fixed. The unique naming convention for the tests was also listed in the defect description. The purpose of this traceability was that when the defect was submitted for fixing the developer could reproduce the exact same test with the same steps and test data.

The test cases were also designed with maintenance of the software in mind. It was expected that they would be reused multiple times and on subsequent builds that only required regression testing. The test cases were subjected to a walkthrough and review with the developers to ensure that the test coverage and test data was adequate. The test cases were subjected to a peer review by the test team to ensure that all areas were sufficiently covered.

While the test cases were designed with intended maximum test coverage it was imperative that they achieve this aim as the existing application was poorly received by customers. To this end they were to be executed on a build with code coverage included. This tool would provide detailed information on the number of code statements and functions executed by the test cases. See 'Project Implementation (Execution and Observation)' later on in this chapter.

Test Data

The test data requirements for testing the project, especially the engineering application were quite complex. The suite of software, while consisting of numerous different applications was interoperable with the information flowing between the controllers to the respective applications. The engineering application was the core application where the information originated and is transmitted to the controllers. Its function was to program the controllers. The ancillary applications are used for the monitoring and maintenance of the controllers in a building.

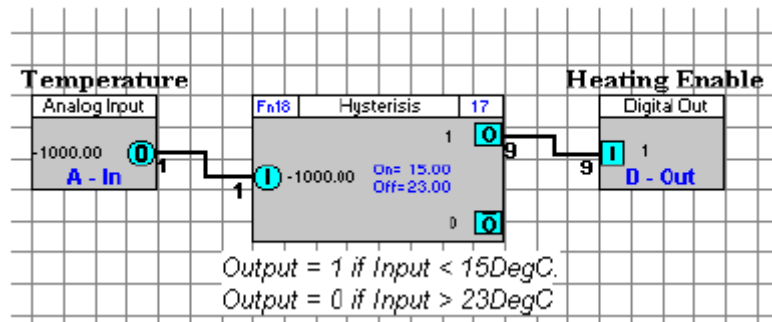


Figure 6.4 temperature control strategy

The controllers could use a maximum of 1024 input and output (I/O) points so each of these blocks had to be tested with connections between the minimum and maximum. The controller also supported 32 Input and Output points, hence the name 'UC32'. The first 8 Input points were configured to accept sensor types of Voltage, Resistance or Current. The remaining 24 I/O points are programmable to operate with either analogue or digital sensors. These programmable points, named 'Uniputs' could also operate as either input or output points. The controller could support a total number of 1024 blocks. Some of the blocks had a maximum amount that could be used in any one strategy. Strategies had to be created that allowed for the connection of each of the 61 strategy blocks, each connection point had to have a connection between the acceptable range of 1 and 1024, with the maximum supported number of blocks in the strategy. Using boundary analysis values of between 1 and 1024 were selected as connection points, with negative tests evaluating point numbers less than 1 and greater than 1024. The strategies also had to test the hardware point configurations with a combination of analogue, digital input and output points. With the different number of input points, strategy modules and combinations of connectivity there were approximately 524,288 possible connections that had to be tested. The strategy modules and points used for positive testing in the strategy are listed below. The different points used in the strategies for testing use the boundary values available and use some of the available combinations possible that yields the most test coverage of all possible combinations.

	analog points	digital points
HW inputs	1,3,5,7	2,4,6,8
Uniputs	9,11,13,15	10,12,14,16
Relay Uniputs		17,18,19,20,21,22,23,24
64 Strategy		
modules	1,5,16,24,32,100,500,1024	2,6,17,31,101,501,1023
Set points	2,6,15,33,99,499,1022	3,7,14,30,98,498,1022

Figure 6.5 strategy module connection point details

Initially single strategies were devised which would test each of the modules. These strategies contained several modules of the same type with each of the different possible connections. These strategies were used to verify that the correct point number was saved in the strategy file which was to be transmitted to the controller.

Single strategies were devised which would test the hardware points in their different configurations. There were 24 hardware points that were tested in each of their Analogue or digital formats as either input or output points. These single strategies were then reused to create larger strategies which incorporated each module up to the maximum of 1024. Equivalent partitioning was used to devise these larger strategies with as many combinations of connection points as possible.

The strategies were also designed with reuse in mind. The test cases were modularised under different application components. There was functionality overlap between these components and that of the test data. The data could be reused or copied and altered for testing the different components. The test data was also scalable; in this regard it could be used for testing each component but also for testing large components together. The integration tests were formed by adding a large number of individual strategies into one large strategy and used on a network of a large number of controllers. This facilitated system testing the applications and controller firmware.

Defect Repository

The defect repository was prepared for the application under test. It had each application component and version entered into the database, so that they could be selected during the entry of each defect entered. Standards were laid down for the entry of the defects so that the reproduction of the defect would be simpler for the developer to facilitate a quick turnaround. The repository allowed the entry of defects with two sets of priorities, one for the impact of the defect on the testing and also for the severity of the defect on customers. The repository also allowed for the history of defects to be recorded as the defect moves between states during its lifecycle, the individual who changed the state was required under the standards to enter in the reasons for the state change.

Testing Execution Management and Team Dynamics

With grey box test case design and thread testing planned there was good team co-operation. The test effort had to be managed in relation to the number and severity of defects. This was organized during team meetings and defect triage meetings. Any outstanding defects were discussed in relation to their impact on the customer and on the test effort. High severity defects were prioritized with development for fixing in subsequent builds. This allowed for early correction of high impact defects.

Version Control

Since the testing was going to be conducted during development in threads of releases, there had to be tight version control. The builds that were released to test were formally handed over with documentation stating the implemented features and what defects were fixed. In turn the cycles of tests that were executed were recorded against the versions of software with the severity and number of defects detected. These metrics defined the state of the quality of the software at any given time. Any versions of software that were above the minimum predefined quality criteria were assessed for release. Any builds that were intended for release to customers had their versions altered so that they could be identified as release builds.

6.1.3 HVAC Project Implementation (Execution and Observation)

The test planning was effective in that test cases and test data were designed well in advance of testing an application; this assisted in reducing the overall time frame of testing. The reuse of test data and the facility of a dedicated test environment also contributed with this reduction of the time for system testing.

The integration and component based testing combined with thread testing improved resource utilisation and efficiency but it brought a lot of test execution tracking problems, where items were tested in previous builds but were subsequently found to be not working in later builds. There were also multiple builds of both software and firmware where different builds supported different features. The version control or configuration management was improved but needed stricter enforcement.

The different test phases that were planned were executed in succession; this assisted with building confidence in the system and highlighting areas that required further attention. The delivery of the applications to test was not punctual. To improve the test execution, the number of builds delivered to test and the number of test cycles executed on them were tracked to improve quality and an emphasis placed on development to ensure that defects were fixed first time around.

Overall the project was completed six months behind schedule, with almost 73% of delivery milestones to test being missed; this in turn led to the delay of test missing their milestones. The over run was in the region of 33% of overall scheduled man days. The controller was released to customers but some features were not implemented. The engineering application (the new version was named ETV6) was also released with some components not modified. It was discovered during testing that certain components of both the firmware and software would need a complete redesign. The root cause analysis of defects revealed that some design solutions were not feasible. If these features were documented properly and a design review held these defects would have been identified much earlier in the project lifecycle. See Chapter 4 'Software quality assurance defect removal' and figure 4.11 'Characteristic distribution of software defect origin' where 33% of defects are injected at the design stage. The likelihood is that these defects would have been detected at a design

review, in figure 4.12 ‘Average defect filtering effectiveness by QA’ 50% of defects at the design stage are detected at a design review. What is startling is that the cost of detecting these defects at the testing stage is approximately 10 times more costly than at a design review, see figure 4.13 ‘Average defect effectiveness cost’.

Despite these setbacks the release was a success for a number of reasons, during testing certain defects that were detected during thread testing were able to be dealt with during development which saved overall development time. There was a 73% delay in development milestones but only a 33% project overrun, in figure 6.4 the number of defects rose sharply, this was the build that QA received that had all the engineering application available for testing. The defects were identified by components and features in components that were not satisfactory were omitted from the release. The release was issued with all priority one and priority two defects fixed but with over a hundred priority 3, 4 and 5 defects still open in the Windows software.

The HVAC Firmware testing was successful from a project perspective. As can be seen in figure 6.4 there were a large number of builds, 13 in total, it wasn’t until build 5.51, or midway through its testing that the engineering application was available to test the firmware, the evidence of the delay in the number of defects can be seen in figure 6.4, where the number of defects detected rose sharply. The advantage of thread testing the firmware was that it was possible to continue testing despite not having all the software available.

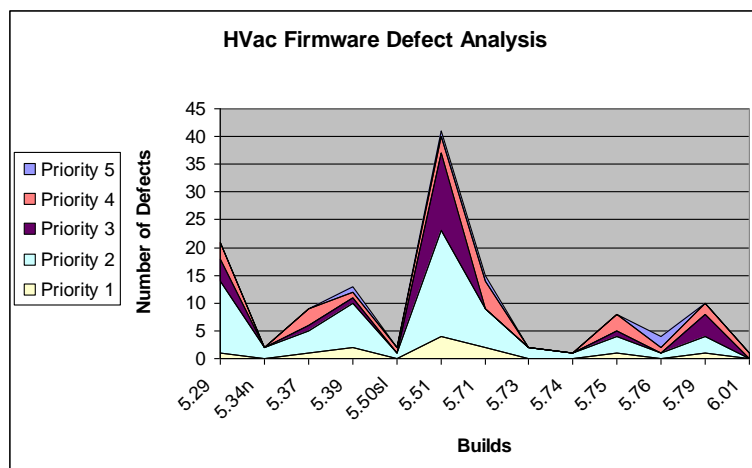


Figure 6.6 HVAC Firmware Defect analysis

In figure 6.7 the tail end of the cumulative defect curve indicates the slowing down of the defect detection rate at the end of testing. This is indicative of test burnout, see chapter 4 ‘Capturing and analysing defect metrics’. In the first 6 months of release, 4 defects were reported by customers. This represents a 3.8% defect escape rate or a 96.2% detection rate which is high. This validates that the testing effort was in the high effort / high outcome bracket for firmware testing, see chapter 4 ‘Capturing and analysing defect metrics’.

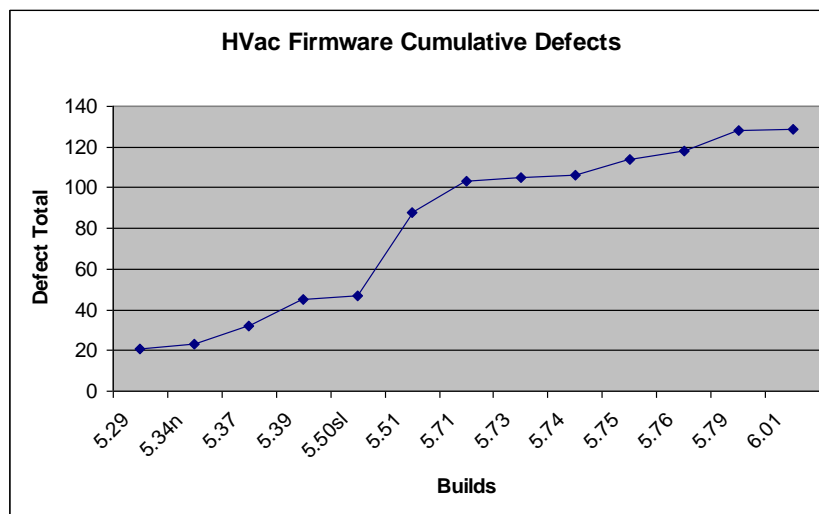


Figure 6.7 HVAC firmware Cumulative defects

The new testing approach demonstrated its effectiveness at reducing the testing time while still yielding a high defect detection rate. The number of different builds that had to be tested was quite demanding on the test team members. The emphasis on quality was to be directed towards the leading edge of the cumulative defect curve and to insist on less defective quality builds. Any defective builds were to be returned to development with a record kept on the number of defects reopened.

The previous release of the engineering application had 1772 defects in total which were tested over 72 different builds, the next version had additional HVAC support and existing functionality re-factored. During the testing of ETV6, some 673 defects were detected over 17 successive builds. The builds and defect severities are displayed in figure 6.9. This represented a five fold reduction in the number of builds required for testing. The duration of the second engineering application project was also significantly reduced with ETV5 taking 960 man days and ETV6 only 430 man

days. The later ETV6 project took only 45% of the time taken for ETV5. This indicates that the efficiency of all testing techniques was beneficial in reducing the overall development time.

During the testing of the engineering application a full regression test was performed on a build which was compiled with code coverage. This build was compiled with a code coverage development tool Devpartner (Compuware corporation, 2005, internet) which provided statement and method coverage of the effectiveness of the test cases. This tool allows the number of source lines of code (and other metrics) to be recorded when the tests are executed; the purpose of this tool is to report back the effectiveness of the test. The test scripts were executed completely using the component based test cases and test data and were completed over a period of 6 days.

The results of the coverage of the test cases were as follows:

Percent of Lines Executed:	61.3
Number of Lines:	68640
Number of Lines Executed:	42070
Number of Lines Not Executed:	26570
Percent of Methods Called:	64.6
Number of Methods:	5403
Number of Methods Called:	3488
Number of Methods Not Called:	1915

Fig 6.8 Test case and code coverage for project 1.

This represents a 65% method call of the entire application over 6 days and 61% statement coverage. The previous test cases took 30 man days to execute. The use of revised test cases, test data and the test environment assisted with an 80% reduction in the effort for regression testing.

The 65% was considered a high level since third party libraries that were compiled in with the source code could not be called by manual testing.

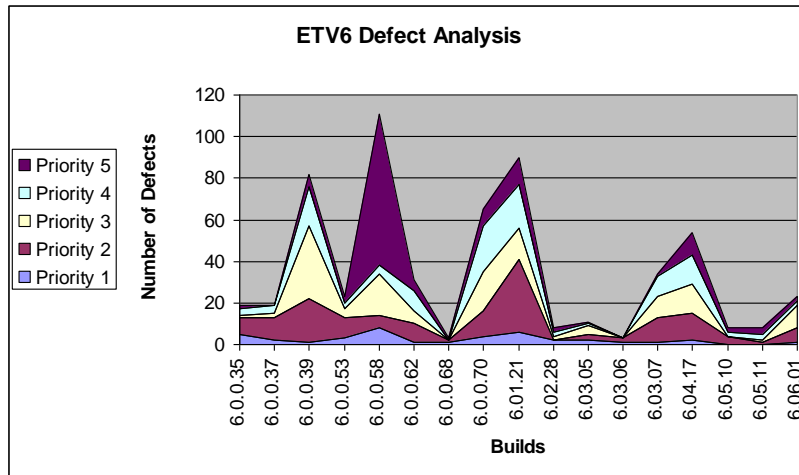


Figure 6.9 Defects per build analyses for the Engineering application ETV6

While the test coverage was considered sufficient, the number of defects that were detected was also of importance. Despite the fact that this application was revised heavily, the number of defects found was a cause for concern. It was good that the test effort detected a high number of defects but it was an indication that the software was of a poor standard.

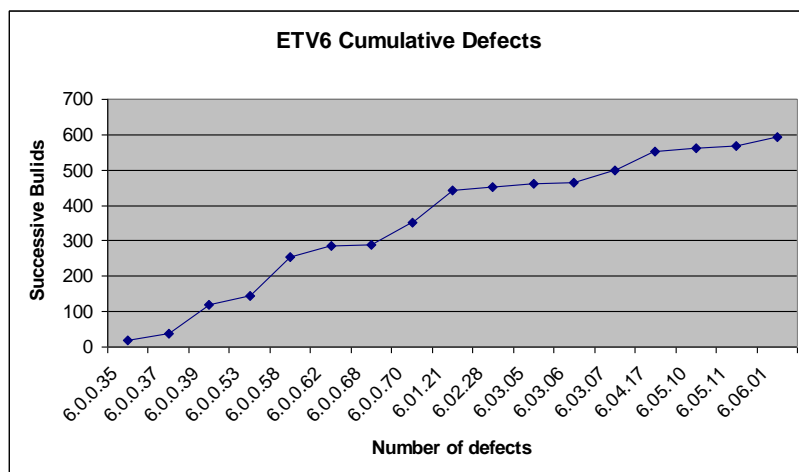


Figure 6.10 Cumulative Number of defects for ETV6

The tail end of the curve in figure 6.10 indicates an upward trend towards an increase in the number of defects despite successive builds.

Analysis of the engineering applications defects per component (in figure 6.11) highlighted the weak areas of the engineering application for further development work. The number of defects detected in the globals and points components reinforced the problematic areas that the customers had experienced with ETV5, the earlier release. The user interface (UI) was particularly weak with many minor defects highlighting a poorly designed application.

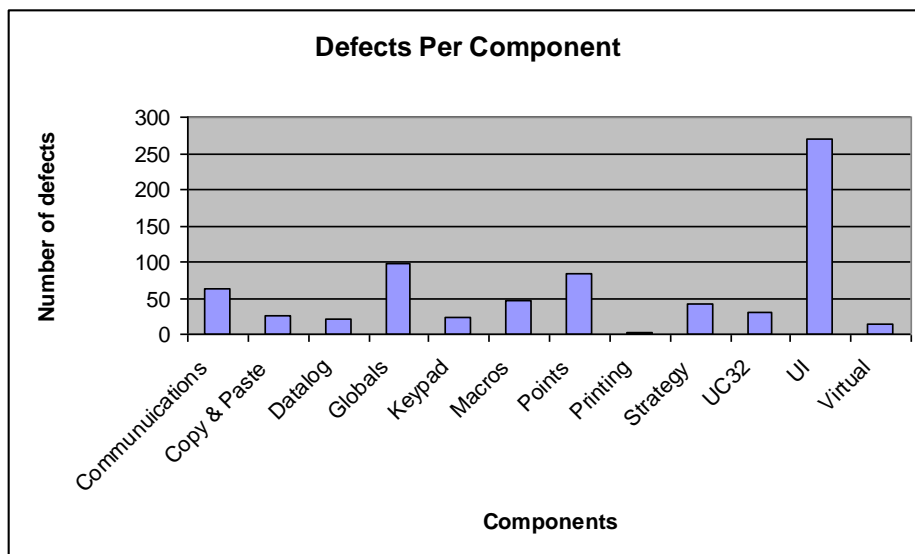


Figure 6.11 ETV6 Defects per component

The component based testing approach allowed most defective areas to be regression tested without a complete retest of all functionality which contributed to a reduction in test time.

The defects for the UI were evaluated and the common causes were used for input into future UI test case design. These tests could be executed on mock up user interfaces to save development costs in future. The most beneficial use of such UI testing would be on prototypes so that any issues would be corrected before costly backend development was undertaken.

The ancillary applications were not as high a risk as the HVAC firmware and ETV6 as they played a supporting role. These applications were tested when time became available during a turnaround in between the firmware and engineering application

testing. After testing any of these applications the higher priority defects were addressed at the project meetings. There were 229 defects found in total for all of the ancillary applications, this is a rate of 9.5/KLOC. When the engineering application and firmware were suitable for release the most stable versions of the ancillary applications were system tested and then install tested before a full release was sent to customers on Beta testing.

6.1.4 HVAC controller project reflection

The benefits for the new practices during the project execution were a high end efficient test effort for both the software and firmware. The phased test plan approach was conducive to tracking the progress of the test effort while also identifying weak areas of the project. The risk based testing approach and the prioritising of defects allowed high risk areas to be completed first and also ensured that in-complete functionality was of a low risk and could be postponed until a later release.

The new test practices had so far proved their benefits in detecting defects and reducing the timeframe for testing. Further improvements would have to be made with co-coordinating the software and firmware builds so that their release would coincide with each other and also to reduce the overall testing time and allow for prompt releases and for deadlines to be met.

The team interaction increased as a result of attendance at peer reviews and project stage meetings and defect triage meetings. The relationships that were developed helped resolve understandings which contributed to increased productivity. The attendance at meetings was sporadic however and was largely dependent on the free time of individuals rather than on a will to attend. The peer review of documents provided additional insight into the application under test and improved the quality of the test cases and data.

The number of builds and resulting testing cycles was proving time-consuming for testing, test automation was considered as a way to alleviate this problem. Either the number of builds or the number of test cycles would have to be reduced to reduce the workload on the test team on future projects. It was estimated that three cycles of testing should be sufficient based on the three large peaks of defects for ETV6, in figure 6.9.

During the testing there were serious design flaws detected in both the firmware and software that should have been detected earlier in the development phase. These flaws

could be averted with improved design and design documentation; the reviews of both of these activities should prevent such defects arising at a late stage in the development cycle in the future. The software User Interface was also quite poor with a significant number of defects; it was proving too costly in terms of development and testing time to maintain. Suggestions were put forward for prototypes to be designed for future projects to assess their suitability and to be evaluated by customers prior to full development.

After the project was released to customers there were an additional 10 medium priority UI defects detected, the customer provided steps to reproduce the defects. There were a number of different steps involved in using the UI to develop a strategy. Following up on the defects that the customers logged, there were different engineering customers following different sequences of steps to develop the same strategy module. Extra effort would need to be placed around either the testing of the different combinations of steps or to have the number of combinations reduced in the UI itself. There were workarounds for the defects, but it was an insight into how differently the UI was being used by different customers. It was an indication of the need to get customer involvement in UI prototypes or user acceptance testing.

6.2 Company X CNET Project Repeat improvements

6.2.1 CNET Project description

The UCX32Net was the next replacement controller. It was a network controller that was responsible for controlling the communication between all HVAC controllers. The UCX32Net project was a smaller project than that of the previous Hvac. The reason for this was that there was less modifications to the ancillary applications. There were 3 applications to be modified with CNet support. It was a project that required 1000 man days effort. There was 6KLOC for the engineering application and 2KLOC for each of the ancillary applications. The CNet had approximately 404 function points. There were three developers and two test engineer on the project, over its duration. Only one of the developers had experience on this work before. The other members of the team were inexperienced. A new web based User interface was planned for the controller, which was to be embedded in an onboard web server.

The engineering application also required further re-factoring to incorporate the CNet protocol support and to complete the components that were not released with the HVAC release (ETV6). These two components were areas of the application that customers reported as defective in the customer survey. The components were the Globals and the strategy screen zooming and printing feature.

6.2.2 CNET Plan

The same test plan layout for the HVAC project was reused as a template for the UCX32Net project with the content updated where necessary. The existing test environment and test data were leveraged for the testing of this controller. There were slight modifications required for the replacement of old controllers for the new CNet controllers but the infrastructure was already in place. The test data that were created for the HVAC firmware testing could be reused without modification.

The test phases identified included the testing of the HVAC firmware, integration, systems testing each application with the new CNet controllers and a regression test of the software with the legacy controllers. The integration testing included serviceability testing each application for correct operation with the HVAC protocol. The system testing included performance testing the CNet controllers for data throughput and the engineering application for multiple strategy operation.

The Hvac test plan was used as a baseline plan, it was estimated that the existing test strategies could be leveraged and that it would take 15 days to execute a complete test of the CNet. This estimate was based on records kept during the previous projects test cycles. Emphasis was placed on the number of builds that were to be given for testing. To assist with reducing the number of test cycles that was required, three iterations of full Integration and System testing cycles were planned, no matter how many builds were given for testing see Figure 6.12 for the project baseline. To ensure that the all defects were fixed, tight control of versions was put in place to ensure that the three cycles could be executed and to cover all test cases and to regression test all defects. The thread testing had been successful on the HVAC controller so it would continue on the CNet project but with all high priority functionality delivered on the first build. The HVAC firmware test cases were used as a template for test case design for the UCX32Net. The defect repository was updated in preparation for the CNet project defects.

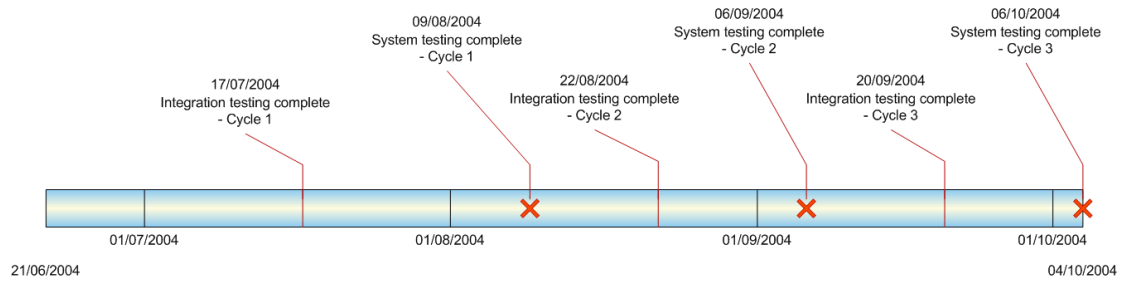


Figure 6.12 CNet project testing timeline with 3 Cycles of testing

The number of defects that were detected in the ETV6 application was a cause for concern and highlighted as a risk. As a precautionary measure the existing code would not be altered in so far as was possible and the new functionality would be developed in a separate UI; a separate windows dynamic linked library ('.dll') which would be called from the existing application.

It was intended that a prototype for the web based user interface was going to be developed and assessed by a selection of customers to verify its use before development was to be completed.

6.2.3 CNET Project Implementation (Execution and Observation)

Project Execution

The test planning was effective in that test cases and test data were designed well in advance of testing the application. The high risk items were tested first and defects were addressed appropriately. The components that were carried over from the HVAC project were implemented and tested thoroughly. The same grey-black box test types were implemented and improved upon in the CNet project which led to improved test estimation and test effort. A prototype for the new web UI was reviewed by customers before the complete UI was developed and embedded on the CNet. This review allowed for functionality that was superfluous to customer's requirements to be omitted from the final UI and for the inclusion of additional functionality which customers desired.

The revised and enhanced test and development practices ensured that the CNet project was completed on time with only two delayed delivery project milestones to test; these did delay the detection of defects in the test cycles, as can be seen in figure 6.13 where it was 6 weeks before a significant number of defects were detected. It was release week +11 that the first significant build was handed over to test. There was one test milestone not met where the final build was released one day late. In figure 6.14 the cumulative defects for the project is a more elongated curve indicating that the time to achieve test burnout was lengthened. The planned three test cycles were completed in cycles of 15, 15 and 10 days respectively. The test case design and existing data and environment had proved beneficial in reducing the test effort and allowing the test effort to bring the project back on schedule.

Note that since 3 builds were anticipated from development, the defect metrics are graphed on a weekly basis to provide results.

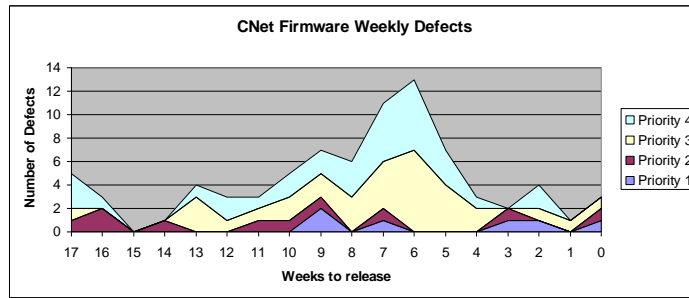


Figure 6.13 CNet defect analysis

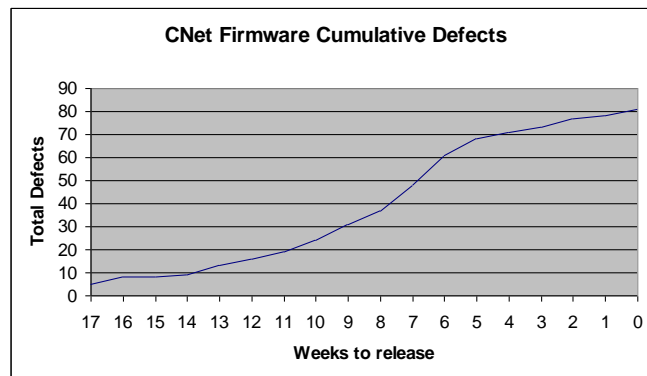


Figure 6.14 CNet Cumulative Defects

The outstanding components in the engineering application were addressed with a new UI. There were only 66 defects detected in UEC6 between the new UI, (see figure 6.15) and its addition to the existing code. There were 7 defects detected by customers after 6 months of release. This was a lower 90% detection ratio than before. The inclusion of the UI as a separate entity was beneficial as that there were not several hundred defects in the HVAC version as a result of code changes.

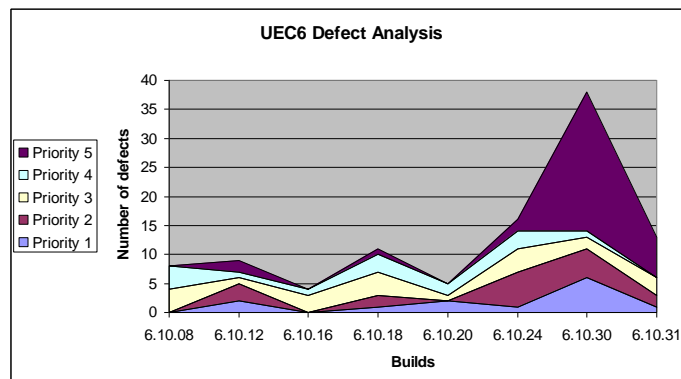


Figure 6.15 UEC6 Defect analysis

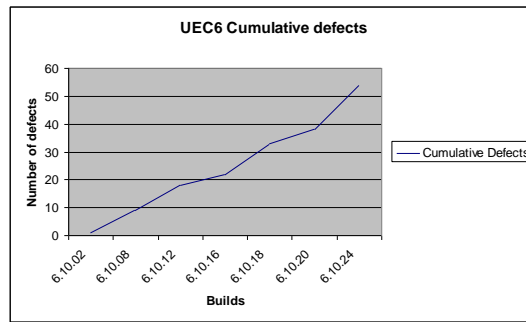


Figure 6.16 UEC6 Cumulative defects

The three peaks of defects in Figure 6.15 for the Engineering Centre are indicative of the three cycles for testing and how full testing can be achieved in this time frame. However in figure 6.15 there is a continued increase in the number of defects detected per build, it is more apparent in the cumulative number of defects in figure 6.11. There is no levelling off at the tail end of the curve. This is indicative of a number of outstanding latent defects in the application. The trend of the graph indicates a continued increase of defects in successive builds. This has been the case for each maintenance release of the engineering application. This is the worst case scenario for testing where there was a high degree of defect injection in the design and code stages of development and where it took minimal test effort to discover a high number of defects. This release required 250 man days which was costly. This cost was a factor in the decision to outsource a replacement application to a low cost development offshore location.

6.2.4 CNET Controller Project Reflection

The successful and timely completion of the CNet project was demonstrative of the improvements in the testing practices of software and firmware for the company. The defect analysis of the CNet testing was indicative of an effective approach to the testing of projects for the company over future projects. The defect analysis for the UEC engineering application however demonstrated that testing alone was not a cost effective solution in providing quality software; it could not alleviate the effects of poor software design. The root cause of the high number of defects per component from the HVAC project and the continued increase of defects for the CNet project justified a rethink of the design of the engineering application.

The data gained and experience gathered during the testing of the HVAC and CNet project would be used to determine the expected quality of future projects. The three iterations of testing software and firmware were found to be an effective benchmark for future project testing requirements.

The defect detection rate for defects per KLOC over the course of the two projects for the engineering centre went from the original version of ETV5 from 29/KLOC to 35/KLOC and again to 35 defects/KLOC. This represents a higher defect detection rate while reducing the overall test time and bringing projects on target. The defect rate is unusually high when compared with the firmware testing defect rate of 3/KLOC. The engineering application rate would indicate a poor quality software application.

Project	Size KLOC	Number of Defects	Defects / KLOC
ETV5	61	1772	29
ETV6	69 (61 + 8)	673	35
UEC6	72 (69 + 3)	66	35
HVAC	22	125	3.5
CNet	26	81	3

Figure 6.17 Projects 1 & 2 quality in terms of KLOC and defects

The statistics were brought to the attention of the board of directors with a recommendation for a redesign of the engineering application; the new design was to be inspected by the test department prior to application development to ensure that software quality could be assessed before costly coding was begun. A design document for the existing application was retrospectively produced in order to gain an understanding of the current applications design.

6.3 Company X – UEC8 Engineering Application

6.3.1 UEC8 Project description

The engineering application was to be re-developed offshore in India. The existing application design was used as a template for the requirements of the new version. The components that were problematic were considered overly complex and more simple requirements were drawn up. Use Cases were written to capture the complex user scenarios that were the cause of a high number of defects in the previous application.

The application was expected to be of a similar size to the existing engineering application with approximately 70KLOC with support for the new HVAC and CNet controllers in addition to the existing controllers and applications. The project team consisted of 12 developers and 4 testers (offshore) over an 11 month period. This was a total resource estimate of 3840 man days. There was one test lead and one development lead in Ireland to evaluate the deliverables from India.

6.3.2 UEC8 Plan

The testing practices, test cases and test data of the previous engineering applications projects were to be used as templates to test the UEC8 application but in the form of User Acceptance testing since the offshore development house had to conduct their own Unit and integration testing. The defect analysis of the earlier previous projects demonstrated that the UI and certain components were quite complex and would need to be delivered to the company in three deliveries.

The three deliverables were representative of the three cycles of testing that were successful in the past. The high risk areas were to be delivered first. Since this was an application heavily dependent on the UI, a prototype was to be delivered to Ireland in addition to the three staged deliveries for assessment and testing. The testing schedule was risk based with high risk areas being tested first.

The static testing of the design documents was also planned in order to shift the focus of quality assessments to earlier in the development lifecycle.

The quality documents and procedures of both companies' were assessed and an interface match was conducted between both development and test life cycles so that there was expectation placed in the contract of the project in terms of quality. A quality plan was outlined and agreed upon. In the quality plan the defect severities were outlined and only a certain number of defect severities were allowed before the application was returned for re - development. The System requirements specification and both the high level and low level design documents were to be static tested before the coding section was to be started. This inspection required the leads in Ireland to inspect the documentation with the intention to gain an understanding of what was proposed from India, and to use their experience to evaluate the strengths and weaknesses of the documentation and to report their findings to management in Ireland and India.

6.3.3 UEC8 Project Implementation (Execution and Observation)

The development process for the offshore contractors followed that of the waterfall development model where a system requirements specifications (SRS) document was written and delivered followed by both high level and low level design documents (HLD – LLD) before a UI prototype and three phased deliveries of the application itself. The contractors own test department was responsible for testing the application before each delivery to the company for UAT. A schedule was developed where each of the projects staged deliverables was broken down by resource and estimated timeframe. The delivery dates were set as milestones in the schedule. Independent testing of the deliverables was scheduled in the company in Ireland.

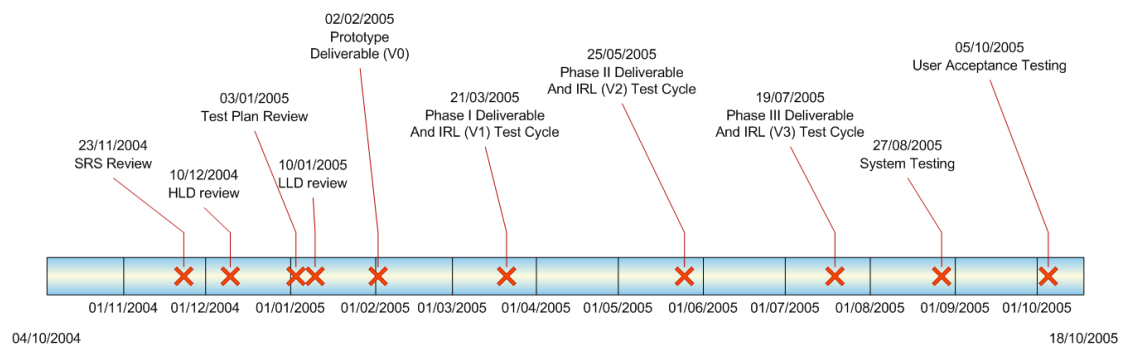


Figure 6.18 UEC8 Project timelines and milestones

The SRS document was the only project milestone that was delivered on target. The high level design documentation was delivered to the company by the contractors three weeks late which was the first milestone missed. The static testing of the design documentation revealed that the design documents did not provide a logical design solution to each of the main components of the application.

The initial UI prototype (phase I delivery) which was delivered was 30 days behind the milestone delivery date. The limited testing that was possible on this prototype revealed that the UI did not offer the functionality that was required for the application. The first application build with functionality available (phase 1 delivery second attempt) which included some of the functionality that was expected as per the schedule in the prototype, failed testing with twenty one defects recorded in testing.

It was not until build (V2.1) that a sufficient level of functionality was present that independent testing could be conducted and the quality plan contract could come into effect in order to reject a build with eighty three defects detected (the contract prevented payment of development until this build was accepted). The prototype functionality and the functionality of the first release were not present in the build until the second release V2.1. With this build the first iteration of functional testing was able to commence. This build and two subsequent builds were rejected on the grounds that the severity and number of defects was below the permitted quality level. There were 34 and 36 defects detected per build. The number of serious defects was increasing with each successive build (4, 5 and 12 respectively). In Fig 6.19 the number of defects increased with each successive build from the contractors. The quality of the software produced was below the expected standard for the software. The test effort was in the low effort and high output bracket where a large number of defects were detected with minimal effort. This is the worst case scenario for software quality. The test effort in the company was user acceptance testing. The Unit, integration and systems testing performed by the contractor was below an acceptable level.

The contractor explained that there was a learning curve associated with the software application and that the quality would improve with subsequent builds and that the contract schedule would need revising, however at the request of the test team; a code review was conducted on the delivered code to provide secondary evidence on the standard of software. The code review corroborated with the findings of the testing, that the standard of code was poor. Based on the test results and code review a decision was made to terminate the contract and cancel the development project. This early termination saved both time and money for the company.

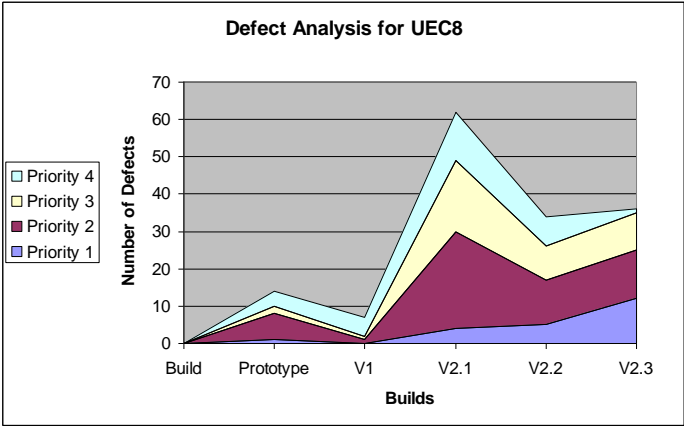


Fig 6.19 UEC8 Defect analysis

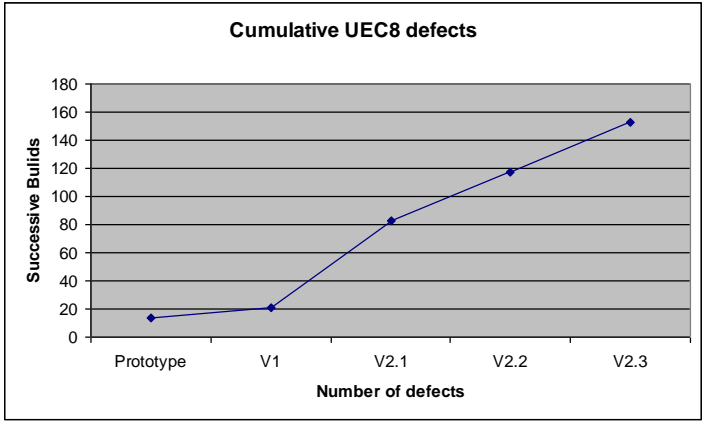


Fig 6.20 UEC8 Cumulative defects

6.3.4 UEC8 project reflection

Independent testing with a formal contract or quality plan provides more leverage for a test team to be effective at voicing their concerns as to the quality of software produced. This forms the basis of entrance or exit criteria for the continuation of the development effort on to the next stage. The standard of the design documents was an immediate concern; as such it reinforces the need for the independent static testing of design documentation before development commences.

The code review solidified the findings of the test effort and should be used early on in the development effort to determine the quality of the code produced before dynamic testing commences.

Summary

As a result of the projects completion, the company launched the new products on the market (except UEC8). The three projects were completed, but with varying amounts of successes and failure. The test process improvements were successful, evidence can be seen with the reduction in test execution time and the artefacts reuse. The defect metrics allowed what the test team experienced to be demonstrated in an effective manner to management. The team interaction was successful; the best example of this was with the successful implementation of thread testing, this required good teamwork and cooperation from all teams. Some areas that require further improvements included the policing of development stage progression, improved reviews at earlier stages would have prevented bad designs to be allowed proceed to coding. The demand by external teams (e.g. sales) on delivery times are sometimes counter productive, the 'rush' to start coding can mean poor quality products are produced which can not be sold. More Quality Assurance involvement at earlier stages can save cost and prevent poor quality products being sold to customer, the UEC 8 project is an example of early QA intervention.

To capture the practices that were successful and to strengthen the weaker practices further, a framework was created. This framework is evaluated in detail in the next chapter.

7 Chapter Seven - Development of a Framework

In chapter six quality problem changes were implemented and evaluated in the company. These practices were evaluated over a number of successive projects. The process was updated with each project until a final model was developed as a framework. The resulting test and QA Framework is intended to be adopted in any development lifecycle model. The framework is wrapped around the five most typical phases of any project lifecycle, see Fig 7.1. These five development phases are broken into two distinct project activities, see Fig 7.2.

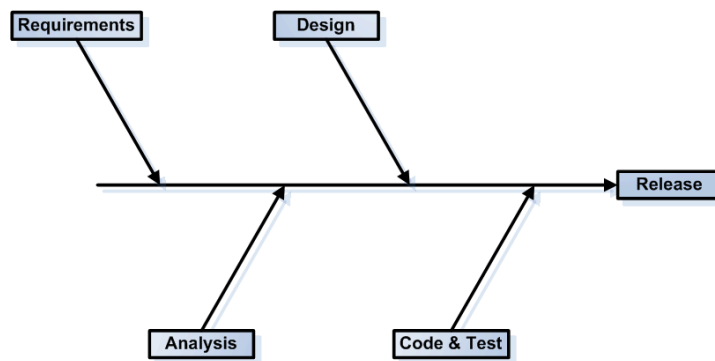


Fig 7.1 five development phases of a project lifecycle.

Quality Assurance Framework

Planning and design
<i>Requirements and systems analysis</i>
<i>System design</i>
Implementation
<i>Coding & test case scripting</i>
<i>Test execution & defect removal</i>
<i>Release & Closure</i>

Fig 7.2 Quality Assurance Framework

The QA framework is split into two distinct phases, the ‘planning and design phase’ and ‘the implementation phase’; these two phases have a number of project stages. In total there are 5 project stages which encompass the entire project lifecycle, see Fig 7.2 for the phase and stage breakdown. During the planning and design phase there is a lot of effort in preparation for the software in terms of user requirements, technical and system specifications. This is a stage in the project that is crucial for getting the project quality on track and is the least expensive stage for removing defects and for preventing further defect injections, 40% of the defects are injected into the project at this stage. See Chapter 4, Fig 4.13 ‘Representative average relative defect removal costs’ and Fig 4.11 ‘Characteristic Distribution of software defects origin’. For this reason it is advantageous for QA to be involved in defect prevention and information gathering prior to testing in the implementation phase. The information gathering assists with the test preparation in terms of test data, test environment and with identifying how to test the solution.

The QA framework has elements of Team Software Process and Rational’s Unified Process where the key project team participants are identified and for each stage there are a number of activities or processes that these key team members have responsibility for. The key members should contribute to the process and produce a number of deliverable artefacts at each process output. The processes are defined in a sequential manner for each project stage so that the output of one process is considered as an input to another. At the output for each process there is a review and a sign off. The purpose of this is to assign responsibility to the key members to prevent defect injection and to ensure defect removal. These items are elements of the Defect Prevention Process and Defect Removal Process (see chapter 4 Defect Prevention Process and Software Quality Assurance Defect Removal Process). The review may take the form of an inspection, peer review or a walk through. There is a sub activity associated with each review to record the metrics of the review and to generate a report to facilitate the documentation of the quality of the software at each phase and also to provide data for the process improvement activities at the end of the project.

The sequence of activities, team participants and documentation associated with each activity is listed for each development phase. Where there is more than one activity or document an associated review must take place and signoff obtained before proceeding on to the next activity. Delays may occur in obtaining sign off in projects but a development phase may not proceed until the next phase without first passing a Go / No go meeting with all domain participants present. This ensures that the quality is assessed and action is taken where necessary. A legend describes the elements to the framework diagram itself in Figure 7.3

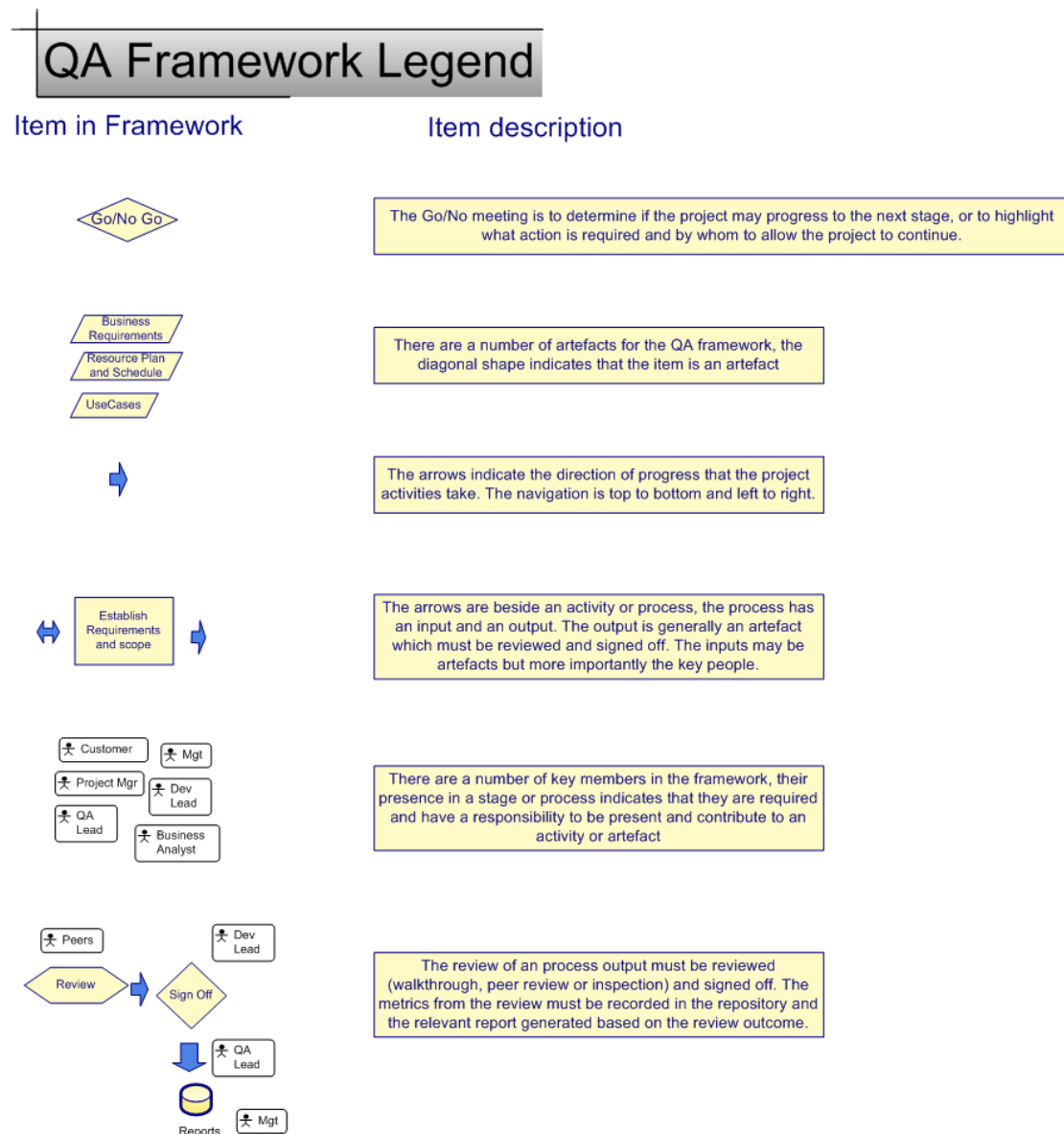


Fig 7.3 The legend for the QA Framework diagrams

7.1 Evolved quality assurance framework

During the course of the evaluation of the process additional documents and modifications to the process were developed. Some enhancements to the process are listed below:

1. The adoption of use cases to document and explain typical customer scenarios.
2. Application prototypes for proof of concept were introduced to facilitate getting sign off of previous documents. A prototype is not a completed software solution but a portion of the solution that indicates the direction that the solution is taking.
3. A template repository for project documentation was also developed to assist with the discovery and retrieval of present and previous project documentation. The repository would be version controlled to assist with configuration management.
4. The addition of a resource plan and schedule for the project so that all team participants regardless of the project stage would have visibility on their inclusion on the project. This facilitated their attendance at review meetings. It also provided a cause and effect indicator if resources were not available to complete certain items of project material. The impact to other departments was more obvious.
5. A separate traceability matrix was created which allowed for the mapping of each user requirement and functional point through analysis, design, coding and test. This matrix was used to supplement the project schedule.
6. A Quality policy that outlines the roles and responsibilities for the project participants in terms of acceptable standards, guidelines and quality criteria for deliverables.
7. A more detailed company technical architecture plan that facilitated discussion at review meetings.
8. Enhancements to the existing URD, SDS, test strategy, test plan documents to cover issues that arose over previous projects.
9. The inclusion of stage meetings (Go – No – Go meeting), as review meetings were not attended by all project participants. It gave an opportunity for a

dependent department to hold the project up pending items to be completed. The benefit was to facilitate outstanding items that were 'lost' during the project to be aired and to have relevant stake holders present to make decisions on the continuation of the project.

10. The inclusion of change control practices to ensure that change requests to the project are recorded and that their impact to the project and participants is assessed before the changes are made. The dissemination of information pertaining to the change requests is handled effectively to reduce the impact on the project.
11. The identification and inclusion into the test process all artefacts of the project lifecycle including those from development for visibility to all project participants.
12. The identification and inclusion of all software and test tools for the project into the process for greater team understanding of deliverables and responsibilities from all team participants. Checklists would be created to ensure that all deliverables were complete before the project would move from one stage to another.
13. The inclusion of a post project review meeting to discuss issues that arose during the project and to address these issues. This review ensures that continuous process improvement is adhered to by making changes as appropriate to the relevant artefacts and processes.

For the purpose of explanation the tools and deliverables that have not been mentioned earlier are listed below.

Defect repository / tool

This is a data store where the details of software defects are recorded. The repository would allow for the status of defects to be identified and for the production of metrics in relation to the defect. E.g. length of time the defect was open and what build it was fixed in.

Reports

This is a report that records information relevant for presentation to management with regards the status of the project for a particular team or stage.

Development tools

This item relates to the software tools that the developers require to fulfil their role on the project. It is included to highlight the responsibility of the developers to ensure that they have the correct tools for the tasks assigned to them. E.g. Code editor and compiler

Development environment

This item relates to the software environment that is necessary for the developer to fulfil their role on the project. It is included to highlight the responsibility of the developers to ensure that they have the correct environment for the tasks assigned to them

Code repository

This is a data store where the software source code is maintained. The repository would allow for the source code to be checked out to individual developers to maintain control over builds.

Test data

This item relates to the generation and maintenance of data that is used during the test process. The data would be versioned and maintained for repeated use. The data would be created to meet with test coverage expectations to ensure as much of the functionality is tested as possible.

Test case repository

This is a data store where the test cases are stored. It is also used to record what the status of the test cases are to report on what tests have been executed, what tests have passed and failed etc. The repository would allow for the status of the software to be assessed at defined intervals, e.g. weekly. The metrics from the test case repository and defect repository should give a good indicator as to what the status of the project software is.

QA environment

This item relates to the software environment that is necessary for the tester to fulfil their role on the project. It is included to highlight the responsibility of the testers to ensure that they have the correct environment for the tasks assigned to them

Build

The build is a version of software that has been released from the code repository. The build may come from development to QA for testing or from QA to customer. The version of the build should be unique so that the contents can be verified with supporting documentation. E.g. handover documents, defect fix reports.

The QA framework is depicted on the next two pages in figures 7.4 and 7.5 respectively. Figure 7.4 depicts the Analysis and design phases and figure 7.5 depicts the Implementation phases.

Planning and Design

QA Framework for the Project Lifecycle

Subtitle

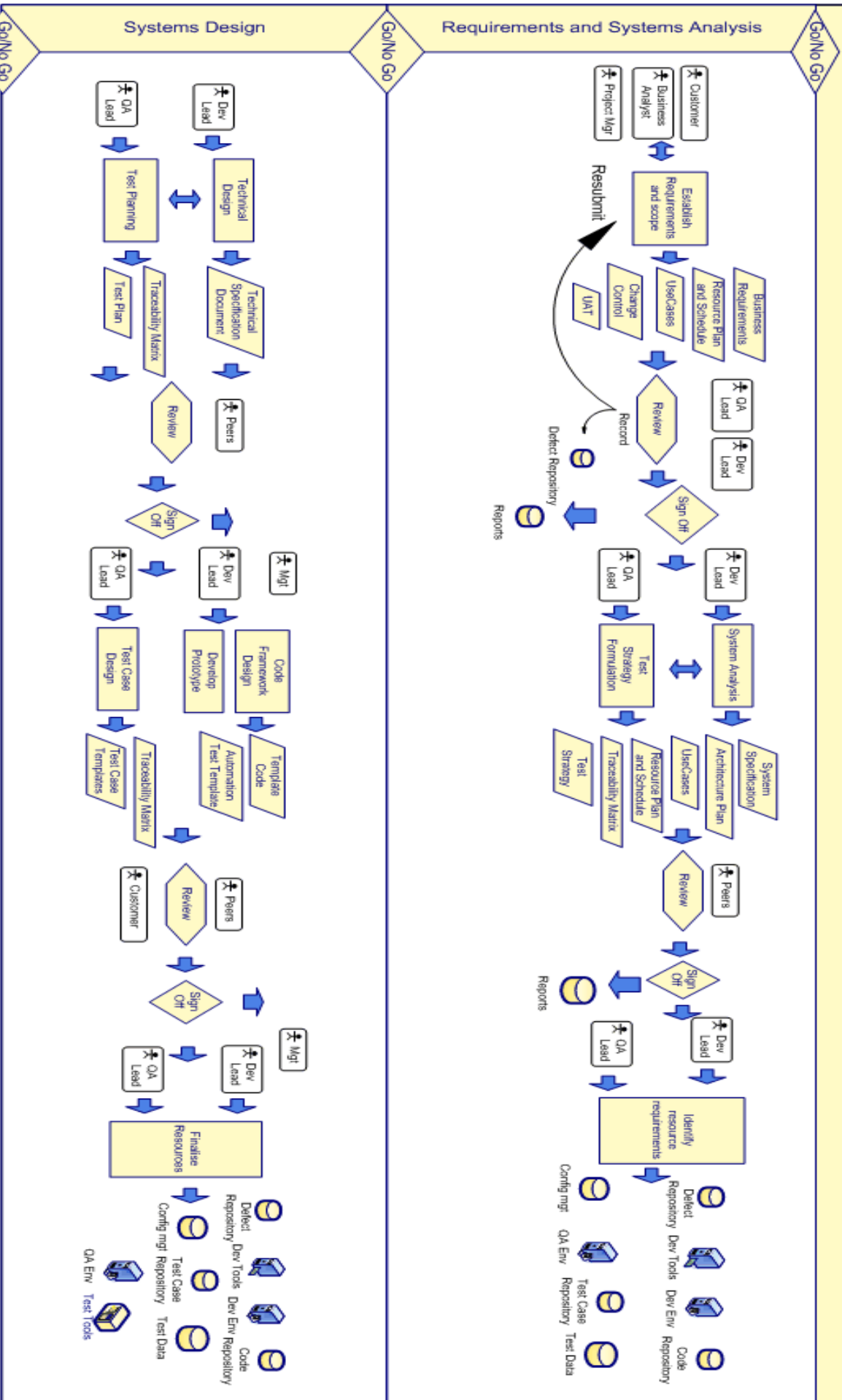


Fig 7.4 the QA Framework Planning and Design Phase

7.2 Secondary Independent Assessment of my proposed Solution

To verify the benefits of the framework, it was deemed necessary to evaluate it in an independent environment in a second company on projects of equal size. The second company agreed to be subjected to QA process improvements and project evaluations over an 18 month period on two projects. The company's Senior President explicitly requested that the company not to be named in this thesis. For this reason the project names and company identity remain absent.

The second company is a large financial institute with a local software site operating in Ireland. The framework was used to design local project process improvements. It was originally targeted on two projects on one of the Irish development teams. However some of the development resources for each project were located off shore. The project documentation *headings* from the earlier projects were re-used as templates for the evaluation.

Both projects were approximately 225 man days in duration successive to each other with a period of 3 months overlap. For project FIIS there were 4 developers in Ireland with one off shore. The project provided a web user interface which interacted with a financial backend database via web services which allowed customers to get updated information on their accounts and to conduct online transactions on their accounts. The application was rated 'AA' in priority with 'A' being the lowest and 'AAA' being the highest rating to be available 24 by 7 with no downtime. There were approximately 52 KLOC and one FP per 63 LOC. The project was developed primarily in Java, JavaScript and Xml with a web type XML UI and http communications with multiple backend Oracle dB's. The second project, B had 3 developers in Ireland and 2 off shore. It was 58KLOC project and one FP approximately per 72 LOC. It was a very similar project for a different financial customer. Both projects had one QA resource in Ireland and one offshore for UAT.

The development team had developed an 'A' rated application before process improvements were conducted. This project CSC, had 73KLOC and one FP per 129

LOC. It was very similar to the project FIIS in its design and execution but had no financial content. There were only 2 developers that were on both project CSC and project FIIS.

The improvements were discussed with other project teams and improvements were then made to other development teams. The results of the project and QA improvements are discussed in the next section.

The improved practices were deduced by conducting action research in one company on two projects and subsequently a third project, and lastly on another two projects in another organisation. In company X there were three projects evaluated against each other and in company Y there was 1 project evaluated with data from a previous project. At the start of each project a plan was devised for improving test practices. These improved practices were carried out during the course of the projects and quantitative and qualitative data recorded during the project's progress. The effects of the practice changes were observed during the project's progress.

When the projects were completed the data was assessed and comparisons made to identify the effectiveness of the practice changes.

7.3 Company Y - Project FIIS Application

To obtain an independent assessment of the Test practices and process to date the practices were evaluated in a different company to assess their effectiveness. The company is an Investment financial institution which develops and maintains its own software. Both projects were developed and evaluated in the same financial software company but with different teams of developers. There was a previous project (CSC) conducted in the company prior to the introduction of the new practices and hence provided a yardstick with which to measure the enhanced test practices.

7.3.1 FIIS Project description

The project that was undertaken to evaluate the test practices was to facilitate customers to get updated information on their accounts and to conduct online transactions. The project code base was approximately 52 KLOC in size with one FP per 63 LOC, see figure 7.6. The project was developed primarily in Java J2EE, with a JavaScript and Xml web type User Interface. The UI communicated with the backend system using Web services which interfaced with multiple Oracle dB's. There were 4 developers in Ireland with one offshore in the US and one QA resource in Ireland and one more in the US for User Acceptance testing. The project was scheduled for 1125 man days. The project was part of a larger overall project but this application was considered independently of the rest of the development effort but with interfaces to the other projects systems.

Project	Number of FP	Size KLOC	FP/ LOC
<i>CSC</i>	<i>565</i>	<i>68</i>	<i>120</i>
<i>FIIS</i>	<i>819</i>	<i>52</i>	<i>63</i>

Fig 7.6 Project size and complexity in terms of FP / LOC

7.3.2 FIIS Plan

The plan for the testing practices on the next two projects was a continuation of the existing testing practices that were successful to date (e.g. test planning, test environment, traceability, test cases, test data, defect tracking, test execution management, team interaction, version control, iterative test cycles, component based testing and risk approach to test cycles) and to evaluate the benefits for the review of all design documentation, independent user acceptance testing and to facilitate regular code reviews to assess the quality of the code early on in the development life cycle.

The development process was based on the Unified Software Development model see chapter 4 ‘Management of the Project life-cycle activities and components’ with elements from the extreme programming see chapter 4 ‘Extreme programming’ and defect prevention process see chapter 4 ‘Defect prevention process’ interleaved. The objective was to embed quality assurance into the development process with emphasis on quality assessments at each stage of the development process. The Team development process was also a factor using the team’s knowledge and experience to its best advantage during reviews and walkthroughs. The lead developer would document and later develop one of the most complex sections of the application. These document and code bases were walked-through with the team and used as templates for the remaining development team. The entrance and exit criteria were determined before each development stage with a team approval required before transitioning to the next stage of development.

The development process is split into two phases, the first phase being to analyse the business requirements and to design a technical solution with both high and low level design documents. The second phase is the actual coding and release of completed builds for testing and deployment. The components to the application were developed into a technical solution in both the system design and technical design documents (SDS & TSD).

Conducting the review of design documentation and the facilitation of code reviews would allow for more knowledge to be obtained on the application and for more effective test case design and test planning. The regular attendance of team members at design reviews was intended to allow project knowledge and domain experience of individuals to be shared with other team members.

The QA and test process lays its foundation with the verification and validation and qualification paradigms where each development stage is verified with its intended efforts against the previous stage deliverables. The test effort is estimated and determined based on the reviewed project requirements and design documentation. The test strategy formulation and test planning follow on from the design reviews where the test types and schedule can be calculated.

The test planning was conducted over two phases, the initial phase was the test strategy where the test approach, test techniques, test cycles, test data, test environment, risks, dependencies, milestones and reports were identified and documented for the project. The application components were identified from the design documents and recorded in a traceability matrix. The purpose of the matrix was for tracking the completion of the code reviews and test case creation

The test strategy was reviewed by all participants before the test plan was documented. This ensured that no item was being overlooked. The test types were identified from the design documents and the unit, integration, system and performance tests were planned. The system and integration testing was again to be executed over three cycles to maximize test coverage. The test data requirements were captured quite early during the test case design. The defects captured during the code reviews were recorded and would be used for analysis of quality.

The actual testing was split between Systems Integration testing and independent user acceptance testing to ensure that anything that was over looked by one test team would be captured by the other team. The Unit test cases were designed with conditional coverage where a tool Junit (Object Mentor, 2006, internet), was adopted for testing the java code before integration testing.

The test cases were designed with positive and negative testing of each component of the application. A test data matrix was compiled with boundary value analysis to cover each of the possible numeric values uses during transactions.

	PIN	
Funds available	Account	Status
\$1	IP	Active
\$1	IP	Inactive
\$1	IP	Presetup
\$1	IP	Brs
\$0	IP	Active
\$-4,4324,876	IP	Inactive
\$0.32165465436	IP	Presetup
\$-0.321	IP	Brs
\$999	SH	Active
\$534	SH	Inactive
\$1287	SH	Presetup
\$6898	SH	Brs
\$-4,4324,876	SH	Active
\$0.32165465436	SH	Inactive
\$-0.321	SH	Presetup
\$0	SH	Brs
\$1,353,654	IP	Active
\$9,545,345,543	IP	Active
\$4,234,643,654,654	IP	Active
\$3,546,234	SH	Active
\$7,654,523,764	SH	Active
\$3,663,234,753	SH	Active
\$1,353,654	IP	Active
\$3,546,234	SH	Active

Fig 7.7 Example test data matrix of account types and fund amounts

There was a lack of domain expertise on the team which was perceived as a risk during the test strategy formulation so a business analyst was added as a resource to the project to provide business domain knowledge that was lacking in the team.

7.3.3 FIIS Project Implementation (Execution and Observation)

The project was completed on time with the testing delayed due to offshore development problems where the application interfaced with other web services. This delay can be seen as a spike in the SIT defects analysis diagram in figure 7.8 during the week of release and again in figure 7.9 in the Cumulative defect diagram. This delay extended the completion of the release by two weeks but had no impact on the overall project. The delay blocked the test case execution and allowed for additional defects to be discovered at a later stage of testing.

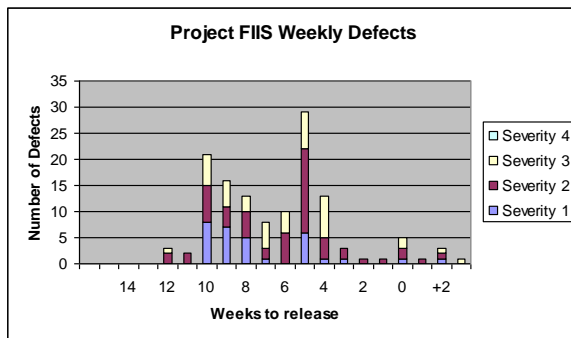


Fig 7.8 Project FIIS weekly defect analysis

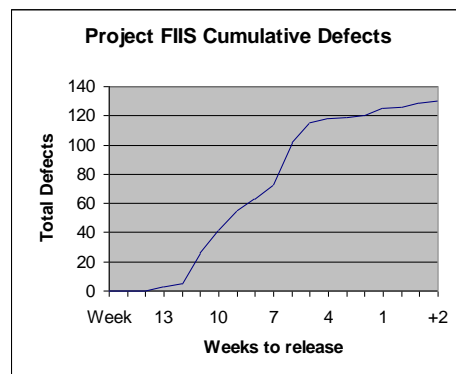


Fig 7.9 Project FIIS Cumulative defects

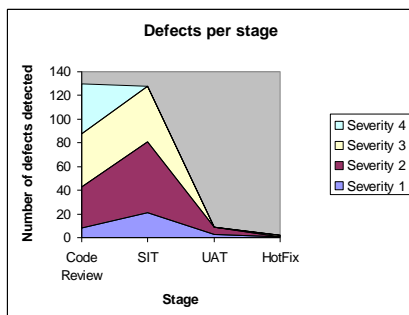


Figure 7.10 Project FIIS defects by stage

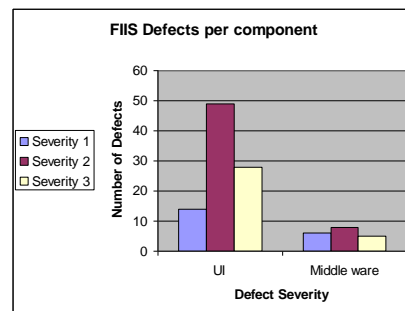


Fig 7.11 Comparison of UI to middleware defect distribution

The review of the design documentation before development prevented any features not being implemented or design flaws detected during testing. The code reviews detected 60% of the total number of defects for the project (see fig 7.10), which had the effect of early detection and removal thereby allowing for the project to be completed on time and without any milestones being missed.

The success of the test practices can be seen through the lack of the number of defects discovered in both UAT and in production (Hot fix). There were 9 defects detected in UAT, this represents an escape ratio of 3%. The purpose of UAT was beneficial with detecting these defects before the application was released to production. After 6 months in production there was 1 further defect detected (Hot fix)

After the project went live and production feedback received a post-mortem meeting was held with the team and topical points from the project discussed.

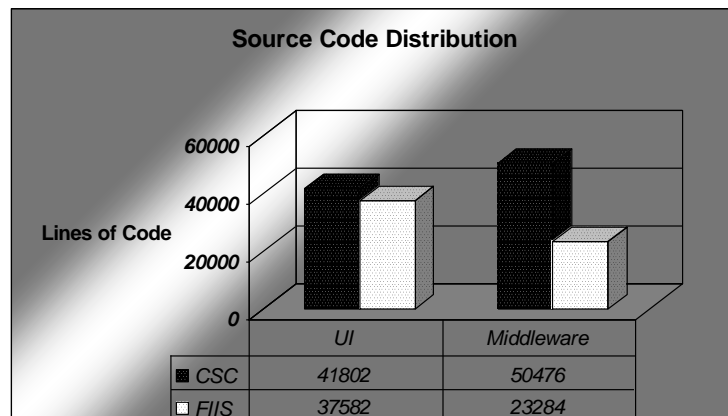


Fig 7.12 comparison of UI to middleware code for CSC and FIIS projects

The Junit testing of the middleware was successful in the reduction of the number of defects that were detected during the testing of the middleware. The number of defects detected in the middleware is approximately 20% of the total defects, where the UI accounts for approximately 80% of the total defects detected, see figure 7.12. The Junit tests were written before the code had been complete so there are no metrics on the number of defects that the Junit testing had detected.

When this project is compared with that of the previous project CSC (the earlier project by the same team of developers with old practices) the difference in the testing practices becomes more obvious. There were 60% more defects in Project CSC when compared to project FIIS. This may be seen as better defect detection in project CSC; however this is not the case when the test effort outcome is assessed where Project CSC delivery milestone was missed by 5 weeks. This indicates a higher effort for higher defect detection. The software quality can be gauged in Figure 7.12 for the number of defects per KLOC for project CSC which was 5.37 where it is 4.5 in project FIIS.

	Project	
	CSC	Project FIIS
KLOC	68	52
Number of defects	365	233
Number of FP	565	820
FP per KLOC	8.3	15.7
Defects / FP	0.64	0.28
Average LOC/FP	120	63
Defects /KLOC	5.36	4.48

Fig 7.13 Project CSC statistics versus project FIIS

7.3.4 FIIS project reflection

The code reviews in particular while identifying a lot of defects were quite popular with the developers which had a mixture of junior and senior developers. The identification of defects at the coding stage contributed to a reduction of defects at a later stage, but the sharing of coding methods was a success in that ideas were shared across the team. This facilitated a 'best of breed' approach to solving issues as they arose during the code reviews. The collaboration between QA and development during the code reviews, documentation reviews and the sharing of the test data allowed for a more positive team dynamic. There was frequent interaction outside of scheduled meetings between team members largely due to the team spirit that had developed. This interaction was useful in solving small blockages in the project progress on an individual basis, which was a contributory factor to the overall efficiency of the team.

The creation of the test data at an early stage of the project allowed more accurate testing of the code with the Junit tests. Each component of the project had Junit tests developed.

7.4 Summary

This chapter describes the foundation for and the framework for test and QA practice improvements. The framework is based on the planning and design phases and the implementation phases of projects. The legend and components of the framework are described in detail. An evaluation of the effectiveness of the framework was conducted in a second software company, company Y. The FIIS project and the quality improvements that were implemented in this project were described. A comparison of a previous project in company Y was made with the FIIS project to highlight the quality improvements.

8 Chapter Eight - Conclusions and Further work

8.1 Conclusion

The aim of this thesis is to investigate the best test and QA practices in industry and to design and evaluate a process for implementing best practices in the software lifecycle of a small to medium enterprise (SME) over successive projects. This thesis was the culmination of over five years of software testing and quality assurance research and practice improvements for software projects in two different SME organisations. To this end the aim of the thesis has been successfully completed. Each of the four objectives in succession led to the resolution of a quality problem in one organisation and for the creation of a framework of proven test and QA practices.

The research into software testing was insightful and of benefit for testing multiple products in different company's. Testing is difficult and requires detailed test plans. These plans must tie the testing approach to the software design and development schedule. This requires careful consideration of the product and demands that resources are prepared in advance of testing. The test plan ideally should be risk based so that it can yield better test benefits where test execution time is limited. Software testing is not sufficient in its own right to ensure that a quality product is realised. There are other quality factors that have to be considered and planned into the project lifecycle. The software test plan should tie in with a project lifecycle process. This project lifecycle process needs to incorporate quality assurance for each deliverable of the project stages to address the quality factors.

Quality assurance from all team members in addition to testers is needed to address all quality factors. The testing of software and QA of each software deliverable requires structure and needs to be an endemic part of a project team. Where each project raises its own difficulties, a process for having QA at each stage of the project is a benefit in surmounting such obstacles. The QA process needs to be incorporated into the project lifecycle with the facility for improvements at project end for feed back into the next project, this continuity of process refinement aids with quality improvements.

If the QA process consists of a combined development and testing process, it is more beneficial in improving the quality of each project phase. With the emphasis of quality in this process, the experience of the QA team can strengthen the project team as a whole in the mindset of Quality Assurance. While the QA process is a combined effort, if the QA team can report independently of the development team, it can be more effective than a dependent team. In addition to an independent QA team, the inclusion of customers in the QA aspect of the project can also have a contribution to improved quality and reduced defects. It is also more effective to have the customers assess quality during different stages of the development cycle. The customers themselves may be included or a body of representatives which can assist with determining the quality assessment of the software.

Software quality metrics are required to track the defects and quality improvements at each stage of the project lifecycle. Graphs of the metrics can be used to plot trends over time of these software quality improvements to assist with the management of the test execution and quality initiative.

8.1.1 Limitations of research

There are limitations in this thesis in respect to the quantitative data used to extrapolate the benefits of the research and also due to the individualistic nature of the project work itself.

Where defect rates and lines of code are determined, they are accumulated over several months of project work and are accurate at the point of their recording from the respective artefacts in which they are stored. There is no allowance made for code that was rewritten a number of times. A simple line code counting application was used to determine as best as possible the number of lines of code for each application.

Every effort was made for the allowance of defects that were opened in error and defects that were assigned an incorrect severity as far as was possible. The man hour and milestone dates are representative of the project target dates and scheduled timeframe. Accurate data was accumulated over the duration of six projects over three years of project work, every attempt was made to keep accurate recordings of each projects respective data irrespective of other projects taking precedence and resources being temporarily reassigned.

The other major limitations to the research are that the projects were carried out by many different individuals; each individual had different work experience and education. The number of lines of code and the number of defects detected are attributed to the work of the individual developers and testers respectively on each project. The exact value of each statistic is determined on a project basis and individual allowances are not represented.

8.1.2 Improvements to practices

The first metric that should be obtained that was not recorded in enough detail would be the number of items detected (design defects) during the review of any design documentation. This could be a peer review of an artefact or the analysis of a document during the test design stage. The cost benefits analysis of the time spent on reviews would be more transparent and support the early inclusion of QA in the projects. This is not the case in most projects.

During the test execution of projects, testing is frequently held up by late delivery of builds or that certain features are not implemented, these test blockages (blocked test cases) should also be recorded as evidence of delays that are not attributed to testing, it would be prudent to include test cases that are blocked and for the duration in which this is the case. Once the testing phase begins, any delays are automatically assumed to be the result of the testing itself. This is frequently not the case.

The additional metrics of design defects and blocked test cases would further support the case for QA reporting on software quality before there is a number of test cases executed and defects reported. It is frequently too late to make significant changes to the software at the test execution phase. The inclusion of metrics at the end of each project phase (the Go / No-Go meetings) would again add weight to any opinions expressed in terms of software quality before proceeding to the next phase. The enthusiasm of developers can often outweigh the pessimism of QA when a project manager's project is under the scrutiny of senior management at meetings.

8.2 Further Work

The areas that could be explored further in relation to this testing process is to be more accurately with the test effort and test outcome. The determination of test effort in terms of the number of resources (man hours) and the test outcome in terms of the number of defects anticipated that a project would produce from both testing and development perspectives based on the number of function points.

Two of the projects were developed off shore, this is an increasingly more frequent approach to software projects and it is an area worth examining further in relation to GSD (Global software development) and the testing of the software developed in this manner. It is increasingly more difficult to co-ordinate a distributed team (virtual team) of developers, testers or business analysts for the purposes of artefact reviews, team meetings and deployment of software builds and releases.

During the testing of some of the projects some of the test cases were automated in conjunction with the maintenance of the test cases. This is a worthwhile activity, but the test automation tools are frequently of the record and playback variety which can extend the project lifecycle. The inclusion of test automation during the development and unit testing of components would be an area that would be worth further pursuit. With Java a test tool 'JUnit' was utilized for the unit testing of the applications in project FIIS. This could be extended further and used in a broader sense for System testing the application in conjunction with the test data for further test coverage and extending the automation of tests. The QA effort while very beneficial for early inclusion in the project perhaps would be best utilized for Test Driven Development (TDD where the test cases are developed before the code is actually written.

The Framework was evaluated in a second company. Further research is necessary on the frameworks adoption across different industry sectors and company's. Only after this research is conducted would the academic community accept its validity and benefits.

9 Appendices, Glossary and Bibliographies

9.1 Appendix A Company X process documentation

The list of Documents as referenced in the thesis are listed below, copies of these documents are at the end of the thesis.

ECR – 0100 Testing Research Plan

Procedure 0029 writing test documents

Procedure 0056 software testing procedure

Work instruction 0032 test script creation

Work instruction 0005 dealing with an incident in released software

Work instruction 0081 use of Bugzilla for defect tracking

Form 0105 software handover form

Form 0123 firmware handover form

Form 0127 SW test report form

9.2 Appendix B – Glossary of terms

Test condition

A test condition is an abstract extraction of the testable requirements from the baseline documents (Requirements, specification, design) A test condition has one or more associated test cases.

Test cases

A test case is a set of test inputs, execution conditions, and expected results developed for a particular test condition to validate specific functionality in the application under test. The percentage of business scope and functionality that is covered by the number of test cases equates to the test coverage.

Test script

A test script is the collection or set of related test cases arranged in the execution flow for testing specific business functionality. A test script must refer to the test conditions covered, the number of test cases that cover these conditions and list the prerequisites for each test condition, the test data required and the instructions for verifying the results.

Software

“Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system” (IEEE)

Software Quality

“The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer” (IEEE quoted in Daniel Galin, 2004, p.24)

Quality control

Quality control is the process by which product quality is compared with applicable standards and that action is carried out if non conformance is detected.

Auditing

Auditing is the inspection/assessment activity that verifies compliance with plans, policies and procedures.

Review Process

A process or meeting during which a work product or set of work products, is presented to project personnel, managers, users, customers, or interested parties for comment or approval.

9.3 Bibliography

- Black, R. (2002). *Managing the testing Process*. USA: Wiley
- Coghlan, D, Brannick T. (2005). *Doing Action Research in your organisation*. IRL.
- Demillo, R.A, McCracken, M. W, Martin R.J, Passafiume, J.F. (1987). *Software Testing and Evaluation*. Addison-Wesley
- Fewster, M, Graham, D. (1999) *Software Test Automation*. London: ACM Press
- Galin, D. (2004). *Software Quality Assurance from theory to implementation*. USA: Pearson Addison-Wesley
- Gao, J.Z, Tsao J. H.-S, Wu, Y. (2003) *Testing and Quality Assurance for Component-Based Software*. USA: Artech House
- Hetzel, B. (1988). *The complete guide to software testing*. USA: John Wiley & Sons Inc
- Jorgensen. P.C. (1995). *Software Testing: A Craftmans approach*. USA: CRC Press
- Kan, S. H. (2004). *Metrics and models in software quality engineering*. USA: Addison –Wesley
- Lewis, W. E. (2005). *Software Testing and Continuous Quality Improvement*. USA: CRC press LLC
- Bain, J. Lee (1978) *Statistical Analysis of Reliability and Life-Testing Models*. USA: Dekker
- Myers, G. J. (2004). *The Art of software testing Second Edition*. USA: Wiley
- Parrington, N. (1989). *Understanding Software Testing*. Halsted Press
- Rational, (2002). *The Rational Unified Process*. USA: Rational
- Raynus, J. (1998). *Software Process Improvement with CMM*. USA: Artech House Publishers
- Sanders, J. Curran, E. (1994). *Software Quality: A framework for success in Software Development and Support*. GB: Addison –Wesley
- Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*, (Texts and Monographs in Computer Science, Springer
- Ernest T Stringer (1996). *Action Research: A Handbook for Practitioners*. USA: Sage

9.4 Web Bibliography

John Mittelstaedt, William Ward, Maya Vijayaraghavan, (2001) Available from: "Location, competition and globalisation: increasing returns and international trade" www.webpages.dcu.ie/~mcdonpi/4-Rahtz-and-McDonagh-eds-2001.pdf, [Accessed 5 Aug 2007]

Irish Software Association Pre-budget submission 2006, ISA executive summary, (2006) Available from: www.software.ie/Sectors/ISADoclib3.nsf, [Accessed 12 September 2007].

CMMI Appraisal Method for Process Improvement , SEI (2001) Available from: (www.sei.cmu.edu/pub/documents/01.reports/01hb001.html) [Accessed 5 September 2007]

Quantitative Software Management, (2005), QSM Function Point Programming Languages Table v3.0, Available from: www.qsm.com/FPGearing.html, [Accessed 12th November 2007].

Compuware corporation, Compuware optimal, (2005), Available from: www.compuware.com/solutions/default.htm, [Accessed September 2005]

Object Mentor, (2006), Available from: www.junit.org, [Accessed October 2005]

Robert N.Charette. Why Software Fails, 2005, Spectrum IEEE, available from: <http://spectrum.ieee.org/sept05/1685>, [Accessed August 2007]

Watts Humphreys. Team Software Process, available from: <http://www.sei.cmu.edu/tsp/>, [Accessed August 2007]

UK Software Metrics Association (1998), available from: <http://www.ukσμα.co.uk>, [Accessed July 2007]

Software Defect reduction
<http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/82.78.pdf>

International Organisation for Standards, ISO available from: www.iso.org/iso/home.htm, [Accessed October 2007]

Rational Unified Process (2002), available from: www.rationale.ie, [Accessed October 2007]

Appendix A – Document Copies

The list of documents as specified in Appendix A are copied in to this thesis. These copies do not contain the Company header and Footer information but do contain the text from the original documents.

ECR-100

Company x Controls Testing Research topics

Author:	Mark Kevitt
----------------	-------------

Approvals:

Reviewed By:	Passed Signature	Date Reviewe	
Issuer	Mark Kevitt	16/12/03	

R&D Manager			Assigned CWP- NONE

Further Distribution:

--

Document Revision History:

Rev.	Date	Details Of Changes	By
1.00	16/12/03	Created	MK

10 Description of Problem(s)

This document outlines the topics that require research for the improvement and modernisation of the testing process in Company x Controls. The reasons for the research are given briefly and a small outline of what is expected from any resultant changes. The topics are subdivided in to relevant sections.

11 Proposed Solution/Change(s)

11.1 Software Lifecycle changes

Implement a more structured version control for both firmware and windows software. New builds to be documented and a process for its release to test.

User requirements documented to be complete before a design/specification document is written. The user requirements to include performance and User Interface requirements.

Specification documents to be approved before coding commences and any changes to the application to be reflected in an updated specification.

A system architecture to be implemented for the explanation of the current software suite and any new applications to be modelled in detail and then added to the overall system architecture.

Testing documents to map to the user requirements and specification document. It is to include risk analysis and user acceptance testing. The test plans are to provide for manual and automation tests.

Determine what is an acceptable beta release standard and what is full release standard.

11.2 Test Process improvement

Testing to be broken in to projects and test team members to be assigned projects on an 'experience' basis.

Test automation to be adopted. The training of the best practises and use of the tool to be implemented.

Bug tracking, Bugzilla database to be backed up on a daily basis. Procedure for the tracking of bugs using Bugzilla to be implemented. Bugzilla emailing problems to be ironed out.

The testing of the a software application to be divided in to three sections/iterations, the first iteration is to automate the tests and to identify as many tickets as possible. The second is to regression test the bug fixes and improve the scripts where necessary. Any tickets found will be documented and metrics calculated. The third regression test will be to verify that the application meets release criteria. After each test section the application will be returned to the developer for the relolution of any tickets.

System testing to be implemented for the release of a CD or a new application which is to be added to the suite. An improved test area to be set up and documented for the system test. It is to include remote modem and TCPIP sites. It should test each application for functionality and performance.

Test to liase with technical support for hand over purposes and for user acceptance testing.

Firmware testing to be addressed, the feasibility of automating the firmware testing using a developed COM interface or other testing tool.

12 Implications of Change(s)

The testing and development process will become more efficient. There will be more transparency of projects.

13 Proposed Implementation

The research will be done on a part time basis and will not impact on the current workings of the test department. The procedures that will be created or modified will be done in accordance with the quality plan.

14 What Tests Will be Required?

N/A

What Documentation Changes Will be Required?

A new procedure will have to be written for the use of Bugzilla.

The work instruction 32 will need to be updated to reflect white box testing of applications. The application and test script should be broken down in to its constituent components.

The procedure 29 will need to be updated to reflect that a test plan and test script can be merged in to one document called a test specification.

The updating of Specification documents WI0022 will need updating to include a model of the application under development.

The creation of a new procedure for the testing of applications in the three iterative process.

Procedure 0029

Owner Dept.	Windows Testing
Modifier:	M Kevitt
Title:	Test Team Leader

Document Revision History

Rev.	Date	Details Of Changes
1.00	9/3/00	Initial Issue
1.01	14/3/00	Change to correct format
2.00	10/11/04	Updated to include template files and Added reference to F-133 Test Doc template and F-134 Test plan template, modified test plan and document procedure to reflect best practices.

1 Purpose

To establish a standard for the creation of test scripts for Software Testing within the Software Testing department.

2 Scope

2.1 This procedure applies to anyone creating scripts in order to test any Company x software.

3 Policy

It is the policy of Company x Controls to create Software Test Scripts in accordance with this procedure.

4 Responsibility.

4.1 It is the responsibility of the Test Team Lead in Company x Controls to ensure that this procedure and the procedures and work instructions it references are adhered to.

4.2 It is the responsibility of the Software Testers to adhere to this procedure and the procedures and work instructions it references.

5 Applicable Documents.

WI-0032 Test Script Creation.

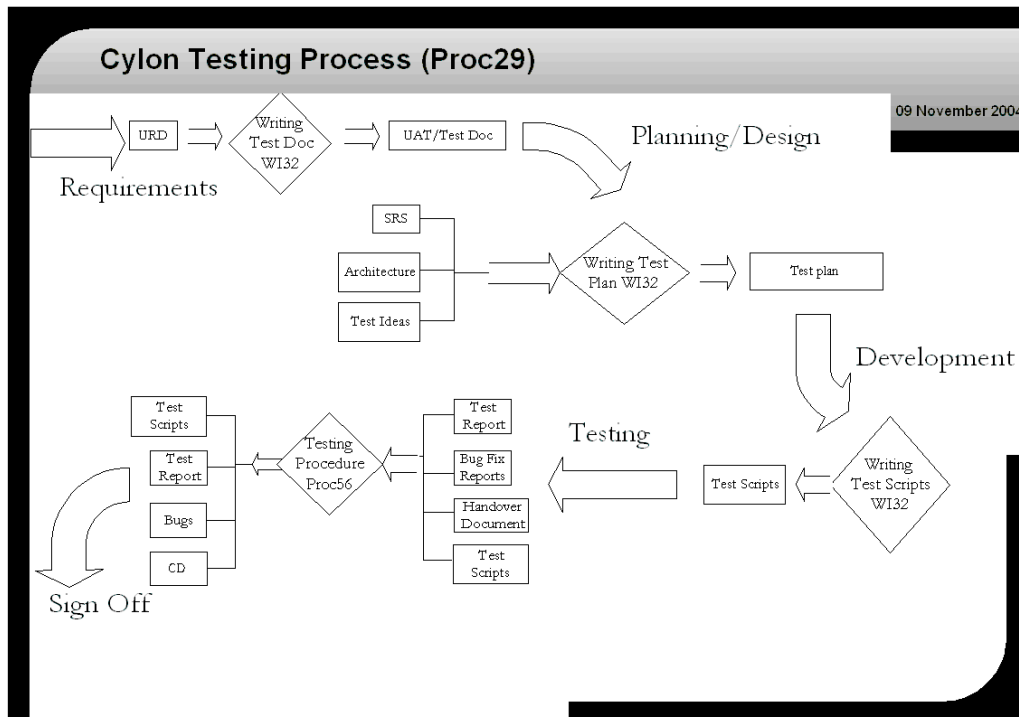
6 Definitions

No definitions applicable

7 General

This Procedure is closely linked with Work Instruction WI-0032. Please read both documents before attempting to create scripts.

8 Procedure



8.1 IMPORTANT:

Before creating any scripts it is important to read all the relevant documentation that refers to the area you are about to create scripts for.

8.2 Before you can move on to create the “Master Test Plan” the “Master Test Document” needs to be signed off by both the programmer, the Test Team Lead and the Customer Services Co-ordinator. Only when this is done can you go ahead and create the “Master Test Plan”. Likewise the “Master Test Plan” must to be signed off by the programmer involved and the Test Team Lead before you can go ahead with the creation of the “Master Test Scripts”.

8.3 Introduction

8.4 This document has been written with the intention of laying out a definitive procedure by which to create the three documents required to create successful and comprehensive test scripts. The first document to be created is a “Master Test Document”, the second being a “Master Test Plan” and the third being the actual “Master Test Scripts”. Below I have broken each of these three down to give a clear outline of the procedure to be used when creating either one or all three of these

documents. Most documentation required to create these documents (User Requirements, Software requirements specification, Design specs etc.) can be found in the project directory. It is good business practice that the tester involved with a project be included with the project at each gate meeting and project meeting to provide input and insight in to the project. This will assist with the project test design and test planning and test document writing. Insist that you are at each meeting pertaining to the project. You can direct any questions at the relevant figurehead. If in doubt consult the QA manager.

8.5 The Master test document (MTD) should be written outlining the purpose of the project and the user acceptance criteria for the project. The Master test document is based on the user requirements document. This document outlines the plans for user acceptance testing (UAT). It should will give an outline of the software that will be tested and inform the reader of whatever purpose or function this particular area serves. Also outlining any new features contained in the software which were not present in previous versions. Here as well should be noted any features in previous release(s) which were not working or working incorrectly. It is the first document to be written as it is used for the initial CWP gate process. It should be written in the format specified in the Master test document/UAT template. The acceptance testing should be carried out by Customer Services after system testing and all testing has been passed. The MTD should have enough information for a user to read and understand the purpose of the new application. It should include enough checklists that the user will have confidents in the application after following the tests in the chcklist. Some checklist criteria are in the sample MTD.

8.6|The Master test plan is written after the master test document has been approved. This document is based on the software requirements specification and the design or architecture plan. The purpose of this document is to plan all testing activity on the project. The plan should be written in accordance with the Test

plan template FXY.Master test plan template.doc

- 8.7** *The Master test plan should only be written after the USR, SRS and Design stages have been complete. Previous projects or applications if they exist should also be researched for outstanding issues or problems so that this information can be factored in to the plan. The stages of the testing have been broken down in the Test plan template. It is imperative that the plan follow the template*
- 8.8** *It can be advantageous to use the numbered points in the “Specification Requirements” document to create your scripts keeping in mind that all points have to be covered (“User Requirements” document is useful for reference purposes). The test cases in the scripts should map to the requirements in order for the tester to easily reference a feature that is being tested.*
- 8.9** *Once scripts are created it is important that their reference details (Number etc.) be entered into the “Requirements Matrix” alongside the function they were created for.*
- 8.10** *If scripts are changed the “Requirements Matrix” will need to be updated to reflect any new scripts that have been added. The same applies if new Requirements are added to the “User Requirements document” if new features are added and new scripts in turn need to be created. These new scripts then will have to be added into the “Requirements Matrix”.*
- 8.11**
- 8.12** *The “Master Test Plan” is an index of exactly what tests are going to take place. Here the “Master Test Plan” creator will break down all the main components and sub components that will need to be tested. The first area to be covered is that all requirements on the “Requirements Matrix” are covered in the scripts. He / She will then create a “Skeleton” or “Index” of all the areas to be tested. Once this Index has been created then the actual names of the tests to be carried out will be given titles underneath their respective heading or*

sub heading. This means that before any test is written in the “Master Test Script” that the script already has a name. Then all that is required is for the steps themselves to be written for each of the named tests.

8.13 ***The component headings then need to be added to Bugzilla after the plan has been approved. This is to facilitate bug tracking when testing commences.***

8.14 ***Master Test Scripts***

These are simply scripts now created from the index you created in the “Master Test Plan”. If while creating the scripts you feel the need to add in new tests along the way (This will happen on a regular basis hopefully if you are testing properly) then remember to update the index in the “Master Test Plan”. For exact steps on how to create test Scripts please see Work Instruction WI – 0032 which can be found in Q-Pulse in the “Documents and Data Control” area.

Procedure 0056

Owner Dept.	Quality
Initiator:	Mark Kevitt
Title:	Test Team Lead

Document Revision History

Rev.	Date	Details Of Changes
1.00	15/04/20 04	Initial Issue

1 Purpose

1.1 To document how software and firmware are tested in the company.

2 Scope

2.1 This procedure applies to all software testing activity in the company regardless of who is doing the testing.

3 Policy

3.1 It is the policy of the company to test software and firmware in accordance with the practices outlined in this document.

4 Responsibility.

4.1 It is the responsibility of the Test Team Lead in The company to ensure that this procedure and the procedures and work instructions it references are adhered to irrespective of who is performing the testing.

4.2 It is the responsibility of the Software Testers and other testing software/firmware to adhere to this procedure and the procedures and work instructions it references.

5 Applicable Documents.

[PROC-0029 Writing Windows Test Documents](#)

[WI-0032 Test Script Creation.](#)

[WI-0005 Dealing with an incident in released UNITRON software interface.](#)

[WI-0081 Use Of Bugzilla](#)

[F-0105 Software Handover Form](#)

[F-0123 Firmware Handover Form](#)

[F-0127 SW Test Report Form](#)

6 Definitions

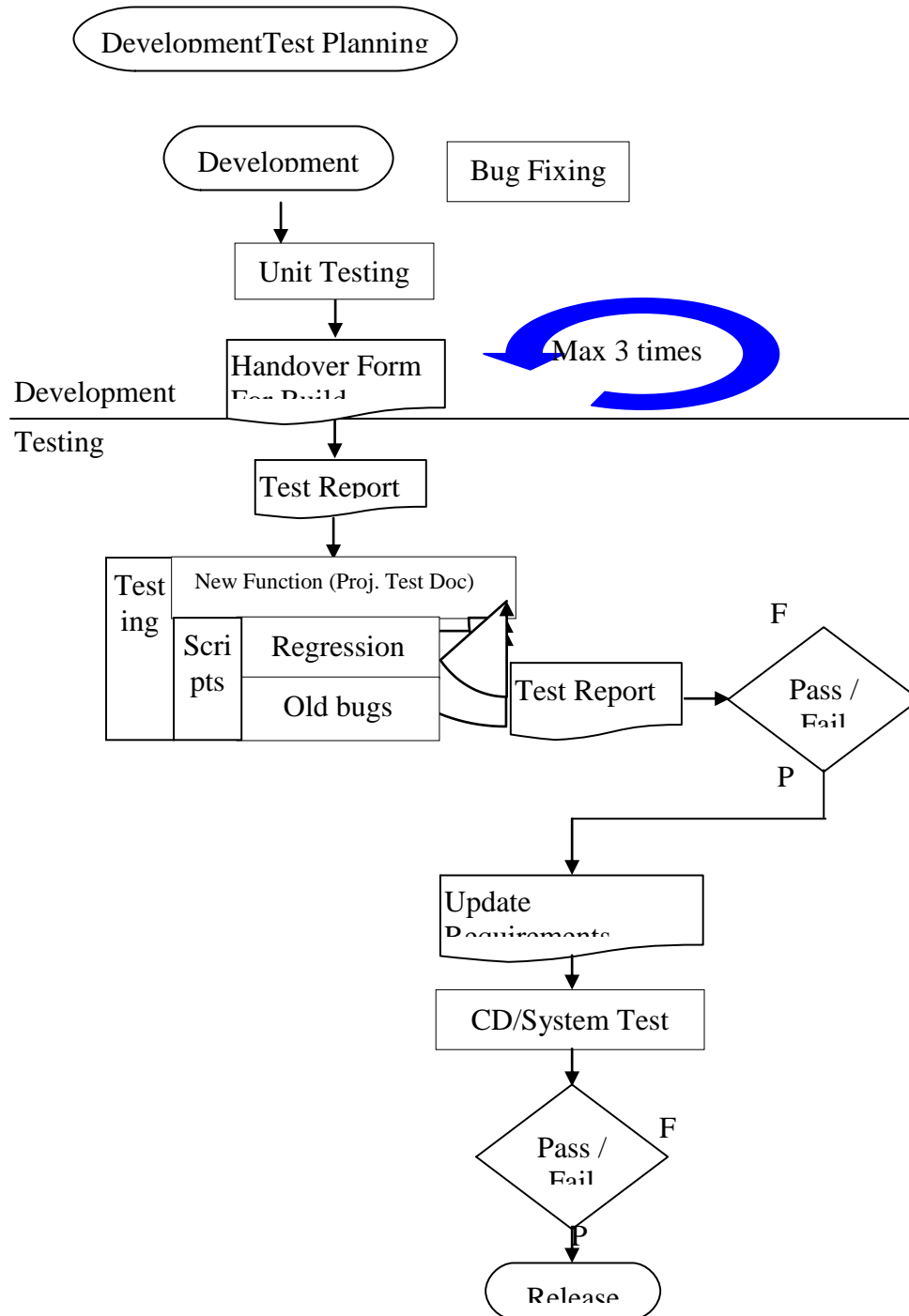
6.1 Bugzilla – defect tracking tool.

7 General

7.1 This Procedure describes the operation of the QA department. It gives an overview of each process, to get a description is detailed of each function. It would be advisable to read all that is listed in the applicable documents section.

8 Procedure

8.1 Test Overview Flowchart:



Goal is maximum of three iterations of testing for a CWP.

Pass One: First pass should be early as possible in development.

Pass Two: Main testing

1.1.1.1 **Pass Three:** Final test, should be

8.2 Test plan

- 8.2.1** The purpose of a test plan is to document and plan what will be tested in a project.
- 8.2.2** The output of the plan will also produce a project schedule that will be used to track the project progress.
- 8.2.3** The test plan is written after the requirements and design documents are approved.
- 8.2.4** The test plan has to be approved by the test team lead and the developer and or project manager on completion, to ensure that everything in the project will be tested sufficiently.
- 8.2.5** The test plan must describe the hardware and software set up necessary to perform the testing.
- 8.2.6** The test plan should also break down the specific areas that will be covered during the testing;. The tests incorporate the requirements of the project and the functionality as described in the design document.
- 8.2.7** Each requirement can be traced from the requirements through the test plan and to the test scripts. The requirements matrix is used for this purpose.

8.3 Test scripts

- 8.3.1 The test scripts have to contain the steps necessary to perform the tests as outlined in the test plan.
- 8.3.2 Each test has a description to facilitate the testers understanding of the actual test. Notes can be added as required to further facilitate the comprehension of the test. The steps necessary for the test are outlined, following the steps are a list of the expected results. This inclusion is to facilitate the testers verification of what was seen during the test to previous known outcomes. A table is provided for the inclusion of the actual results. This table must include provision for a defect number, the signature of the tester, a description of the actual results and a pass/fail field. The creation of test scripts is outlined in [WI-0032 Test Script Creation](#) and [PROC-0029, Writing Windows Test Documents](#).

8.4 Handover

- 8.4.1 The respective handover form ([F-0105](#) for SW/[F-0123](#) for FW) is completed by the developer and handed over to the tester. The form is handed over with the (or the location of) required software and or firmware for the testing.
- 8.4.2 Only when the tester is satisfied has signed to accept the handover, will testing commence, this may involve performing a 'smoke test' to verify minimum quality level . The purpose of the form is to ensure that the coding and administration required of the developer is complete before the testing begins. All of the changes made since the last test on a particular build are listed to assist the tester with their knowledge of the project before testing commences. For example the run log is appended so that each code change since the last tested build can be viewed. The list of fixed bugs can also be added. This assists the tester with the test report and with preparation of the testing.

8.5 Bug Tracking

- 8.5.1 While following the execution of the test scripts the testers may encounter bugs. The tester will log bugs in Bugzilla for the application that they are testing. They will also fail the test in the scripts and log the bug number in the test script. They will follow [WI-0081 Use of Bugzilla](#) for its correct use. Bugzilla will email the relevant development manager about the bug.
- 8.5.2 The bug lifecycle (as per WI-0081) will be followed by the development manager who will assign a developer if

appropriate to fix the bug. During a later test cycle when the tester is testing fixed bugs they will close or reopen the bug as appropriate.

8.6 Project Testing cycles

- 8.6.1** The purpose of having testing cycles is to maximise the productivity of the testing activity and measure the quality level of each build. Each phase of testing is planned and results documented on [F-0127, SW Test Report Form](#) which will have been assigned a test no. from the SW Test Report Log.
- 8.6.2** There can be any number of testing cycles but the optimum is three. On handover of the project the tester will complete the first section of F-0127, SW Test Report Form.
- 8.6.3** The initial test cycle will be for complete test script completion with all defects reported. The cycle should be the longest and will allow the development effort to fix the bugs as they are raised.
- 8.6.4** On the completion of the initial cycle the tester should update the test scripts and test plan if necessary and the also the requirements matrix. On completion of the test the tester will complete the second section of the test report.
- 8.6.5** The development team, will optimally have the next release ready with most or all of the bugs fixed. The tester will complete a test report for the next build based on the handover form. If the bugs are fixed in accordance with the bug fixing procedures then the bug list and related areas as per the bug fix report should suffice in test coverage of the project. The list of bugs fixed will be tested with each affected area in accordance with WI-0005. On completion of the test the tester will complete the second section of the test report.
- 8.6.6** The third test cycle will be followed in the same manner. Should new requirements be introduced or the project be altered substantially then the test plan and scripts will need to be updated appropriately and the next test cycle should be performed as an initial test cycle and continue the cycles in sequence.
- 8.6.7** When a project is ready for release on Beta or full release the tester follows [WI-0069, Changing a Program from Untested to Release](#), and changes the status of the application to Beta or

Release as appropriate.

8.7 Regression testing

- 8.7.1 Regression testing is performed when there have been substantial code changes to a product. A proportion or all of the appropriate test scripts are executed, this is agreed in advance of testing starting..**
- 8.7.2 The handover form will list the changes to the project. The Test Report will be filled out and agreed in accordance to the amount of code that has changed. Certain sections of the scripts may be omitted from a test if that code has remain unaltered and the code changes that were made have no effect on that section of code, again as agreed on the Test Report. Regression testing follows the project testing cycles.**

8.8 Reports

- 8.8.1 The tester will produce weekly reports highlighting any major problems that are preventing the progress of the project.**

The team lead will produce Statistics from the bug tracking database to report on the progress of the project.

8.9 CD/System Testing

- 8.9.1 The CD is tested as an install program. There are purpose-written test scripts that are executed for the testing of a CD release. A list of the applications that are to be installed are provided for the tester for comparison purposes. In accordance with the test script each application version is tested.**
- 8.9.2 Any bugs that the CD addresses are also tested in accordance with WI –0005. The Q-Pulse database and Bugzilla are checked to ensure that all bugs required have passed test before the CD install is tested. A brief system test is also performed on the system after the install.**

Work Instruction 0032

Owner Dept.	Windows Testing
Initiator	*****
Title	Test Team Leader

Document Revision History

Rev.	Date	Details Of Changes
1.00	9/3/00	Initial Issue
1.01	14/3/00	Change to correct format
1.03	2/6/00	Updated after changes from test team
1.04	6/6/00	Further format update

1. **Purpose**

To establish a standard for the creation of test scripts for Software Testing within the Software Testing department.

2. **Scope**

This work instruction applies to anyone creating scripts in order to test any Company x software.

3. **Policy**

It is the policy of Company x Controls Ltd. to create Software Test Scripts in accordance with this work instruction.

4. **Responsibility**

It is the responsibility of the Test Team Lead in Company x Controls to ensure that this work instruction and the work instructions and procedures it references are adhered to. It is the responsibility of the Testers to adhere to this work instruction and the work instructions and procedures it references.

5 **Applicable Documents.**

PROC – 0029 Software Test Script Creation

F - 0076 Test Script Template

5. **General**

This Work Instruction is closely linked with Procedure PROC - 0029. Please read both documents before attempting to create scripts.

1 **SCRIPT CREATION:**

1.1 IMPORTANT:

1.1.1 Before creating any scripts it is important to read all the relevant documentation in the relevant project folders that refers to the area you are about to create scripts for.

1.1.2 Before you can move on to create the “Master Test Plan” the “Master Test Document” needs to be signed off by both the programmer and the Test Team Lead. Only when this is done can you go ahead and create the “Master Test Plan”. Likewise the “Master Test Plan” must to be signed off by the programmer involved and the Test Team Lead before you can go ahead with the creation of the “Master Test Scripts”.

2 If you are about to create the “ **Master Test Plan**” for “**Microsoft’s Outlook 97**”, the first thing you would need to do would be to break down Outlook 97 into its main areas for testing.

2.1 These would be:

2.1.1 Inbox

2.1.2 Calendar

2.1.3 Contacts

2.1.4 Tasks

2.1.5 Journal

2.1.6 Notes

3 Once you have these main areas you need to select one of these and further break this main areas down into it’s respective sub sections. We will select “**Inbox**” for our Example.

3.1 Our sub sections for “Inbox” will be

- 3.1.1 Menus**
- 3.1.2 Icon Bar**
- 3.1.3 Tool Bar**
- 3.1.4 Outlook Icon view side bar**
- 3.1.5 Mail Field (Where mails are actually visually represented)**
- 3.1.6 Window Title bar**

- 4 Once you have done this you need to select one of these sub headings. We will select “**Menus**” for this example and break this down into its sub areas.

4.1 Our sub sections for “Menus” are

- 4.1.1 File**
- 4.1.2 Edit**
- 4.1.3 View**
- 4.1.4 Go**
- 4.1.5 Tools**
- 4.1.6 Compose**

- 5 Now once we have broken down our sections into in to sizeable testing chunks we now begin the final step for the “**Master Test Plan**” and that is to give each separate script a name. We will use the “**File**” menu as an example.

5.1 So under “File” I will have the following names of tests to

be written.

- *File/New* – Correct modules appear

- *File/New* – Correct number of options
- *File/New* – Hotkeys present and functioning
- *File/New* – Icons present and functioning
- *File/New* – Shortcut keys present and functioning
- *File/New* – Text is correct format

- 6 So if my **“Master Test Plan”** is testing Microsoft’s Outlook97 the name of the **“Master Test Plan”** will be **“Outlook 97 - Master Test Plan”**. Your heading below this will be named **“Inbox - Menus”**. The heading underneath this will then be **“File Menu”**. Directly underneath this you will have all the names of all the tests you will end up writing in your **“Master Test Script”**, please see above for example test names.
- 7 Also note that I have used Outlook 97 for example purposes only. Sometimes an area will involve the testing of an actual procedure or complicated action. This can still however be carried out following the above **“Master Test Plan”** document.

8 Master Test Scripts

8.1 Once you have both the “Master Test Document” and “Master Test Plan” completed the scripts themselves should be fairly straightforward and uncomplicated to write.

8.2 The test scripts themselves should also contain an area to fill in your results from each test undertaken. The very first page of your scripts should clearly show how many tests Passed/Failed. If one script has failed then the whole application has failed to pass the testing procedure and it should be clearly marked so on the front page. Each script also needs to be signed off by the programmer working on the application that the scripts are created for.

8.3 Also on the front page is “Level of Knowledge required”. This is filled in by the original person who created the scripts with the aid of the Test team lead. Basically this outlines any technical knowledge (both hardware / software /firmware) required to carry out the scripts.

EXAMPLE:

Tester Name: *A Tester*

Date: *22/2/00*

Level of Knowledge Required:

Application Name: *Calarms (Alarm Handler)*

Build & Version No: *Build 5.00 Version 1*

Passed: *199*

Failed: *1*

Operating System:

Total Script Status: *FAILED* (Here should be a pass or fail)

QA Engineer Signature:

Programmer Signature:

QA Lead Signature:

8.4 *The names for all the tests to be created are present in the “Master Test Plan” and the only thing left to do is create the steps to test the specified area.*

8.5 *Tested scripts can only be filed and considered completed once signed off by both the test team lead and programmer involved.*

8.6 *IMPORTANT: Please note that all incidents discovered in software that is not in general distribution (Meaning that it is not freely available to all our customers) needs to be logged in a separate Excel worksheet and not into Q-Pulse. Once you have verified that there are no other lists in existence (Possibly from old tests that have been carried out) enter your own list in VSS (Visual Source Safe). Go to the “Test Area” section and enter the Excel worksheet under the correct application folder.*

Work instruction 05

Owner Dept.	Windows Development
Initiator	EK
Title	Windows Development

Document Revision History

Rev.	Date	Details Of Changes	By
0	5/10/99	Initial Issue	EK
1	12/11/99	Changed to accommodate Q-Pulse	EK
2	17/11/99	Title changed	EK
3	20/12/99	Step 6 made more explicit. Clarification made to "Identification". Bug changed to Incident	EK
4.00	7/5/2002	"Notes on files here" dropped - all details are in the run logs Technical support will confirm bug fixed with customer	EK
4.01	7/5/2002	Added a note that this applies to feature requests as well as bugs WNNEWVER is now __WNNEWVER	EK
4.02	15/5/2002	Tester changes "Untested" to "Release" before giving to technical support Technical writer adds technical support to PCD for the web	EK
4.03	25/07/2002	Added point where the status is changed from 'fixed' to 'passed test' for tech support	MK
5.00	3/6/2003	Changed location of software to R:\Windows Group and defined that urgent software can be issued by ECO	EK
5.01	22/7/2003	DLL is to be stored in the same directory as the EXE	EK
5.02	23/2/2004	Changed 'Q_Pulse' to Appropriate bug tracking since there are now 2 DB's (Q_Pulse & Bugzilla). Added point 12 'For all incidents in Bugzilla the tester will change the status to closed.' Amended Pt 6 where a handover form will be used.	MK

Note that this work instruction also applies to feature requests not covered by a separate ECR or CWP

1 Method

- 1.1 Engineer fixes incident following work instruction WI-0006 - Fixing an Incident in Release Unitron Software.**
- 1.2 Engineer completes the "Incident Fix Report" (F-0062) part 1 and gets it signed off by the Windows Development Manager or a senior engineer assigned by him.**
- 1.3 Engineer updates Appropriate bug tracking database by marking the incident as fixed in the "Status" field, assigning the incident to the Test Lead (not applicable for Bugzilla) and describing the fix in the Follow Up/Corrective Action field or the Comments field in Bugzilla. At a minimum this should include description of the fix, a reference to the bug fix report and the build number of the fixed executable and dll if appropriate.**
- 1.4 Engineer puts the executable and the DLL that it works with into "R:\Windows Group\Program name\Version.Build" – e.g. R:\Windows Group\CCPager\5.40B06**
- 1.5 The Appropriate bug tracking DB will automatically email the incident number to the test lead so that he can assign it to a tester.**
- 1.6 The engineer places the signed incident fix report in the application directory in the filing cabinet. At a later date the handover form F-0105 will be given to the test dept with the bug fix reports describing each code change to the application.**
- 1.7 If the customer needs this fix very urgently (as defined by technical support), the programmer issues an ECO and copies the software into S:__WNNEWVER\Release name\ECOed\Alpha**
- 1.8 Tester confirms that the incident is fixed following work instruction WI-0033, "Incident Verification in UNITRON software interface" and fills in the incident fix report part**

2 which is then signed by the test lead.

- 1.9 The tester changes the comment field in the version details to "Release" following work instruction WI-0069.**
- 1.10 In the case of incidents raised by technical support or external customers in Q_Pulse:**
- 1.11 The tester updates the Q-Pulse database by assigning the incident to the technical support person who entered it (if you are not sure, assign it to the technical support coordinator).**
- 1.12 Tester updates the status from 'fixed' to 'passedtest', and saves the change.**
- 1.13 If the customer needs this fix urgently (as defined by technical support), the programmer issues an ECO and copies the software into S:__WNNEWVER\Release name\ECOed\Beta**
- 1.14 Technical support will verify the fix with the customer as per work instruction WI-0070 and when it has been verified it will be reassigned to the Windows Development manager.**
- 1.15 For all other incidents in Q-Pulse:**
- 1.16 The tester updates the database by assigning the incident to the Windows Development Manager. If at this point the "Approver" field is empty it should be set to the person who entered the incident.**
- 1.17 For all incidents in Bugzilla the tester will change the status to closed.**
- 1.18 The appropriate bug tracking db will automatically email the incident number to the test lead with a note indicating that it has been closed or reassigned to technical support. It should also automatically notify the person who raised the incident if their name is in the "appropriate" field .**
- 1.19 When the fixed incident has been reassigned to the Windows Development Manager he will assign a**

programmer to issue an ECO. When the ECO has been issued, the programmer the software into S:__WNNEWVER\Release name\ECOed\Release and sends an email to the technical writer with the details of the fix and a request that the new software is put in the "Software Updates" section of the Company x web site. These details will include whether or not a technical bulletin and/or manual changes are required.

1.20 When the technical writer puts a fix on the web, the technical support coordinator and marketing will be on the signoff list for the product control document so that they can advise all other customers of patch existence.

Work Instruction 0081

Owner Dept.	Quality
Initiator:	Mark Kevitt
Title:	Test Team Leader

Document Revision History

Rev.	Date	Details Of Changes
1.00	11/2/04	Initial Issue
1.01	24/2/04	Made changes as commented by EK, changed the lifecycle to reflect that the tester assigns bugs to the relevant manager. Added in sections for reassigning a bug and for changing a bug to fixed.

Document Shortcuts:

[Bugzilla](#)

[Login](#)

[Mail settings](#)

[Entering bugs](#)

[Querying existing bugs](#)

[Changing the status of a bug to fixed.](#)

[ReassigningABug](#)

[Bug Status Cycle](#)

2 Objective

To establish uniform practice for the operation of the Bugzilla bug tracking software.

3 Frequency

This procedure applies to anyone who uses Bugzilla and should be referenced when any confusion arises through its use.

4 Applicable Documents

WI-0064 software test acceptance of non released software.

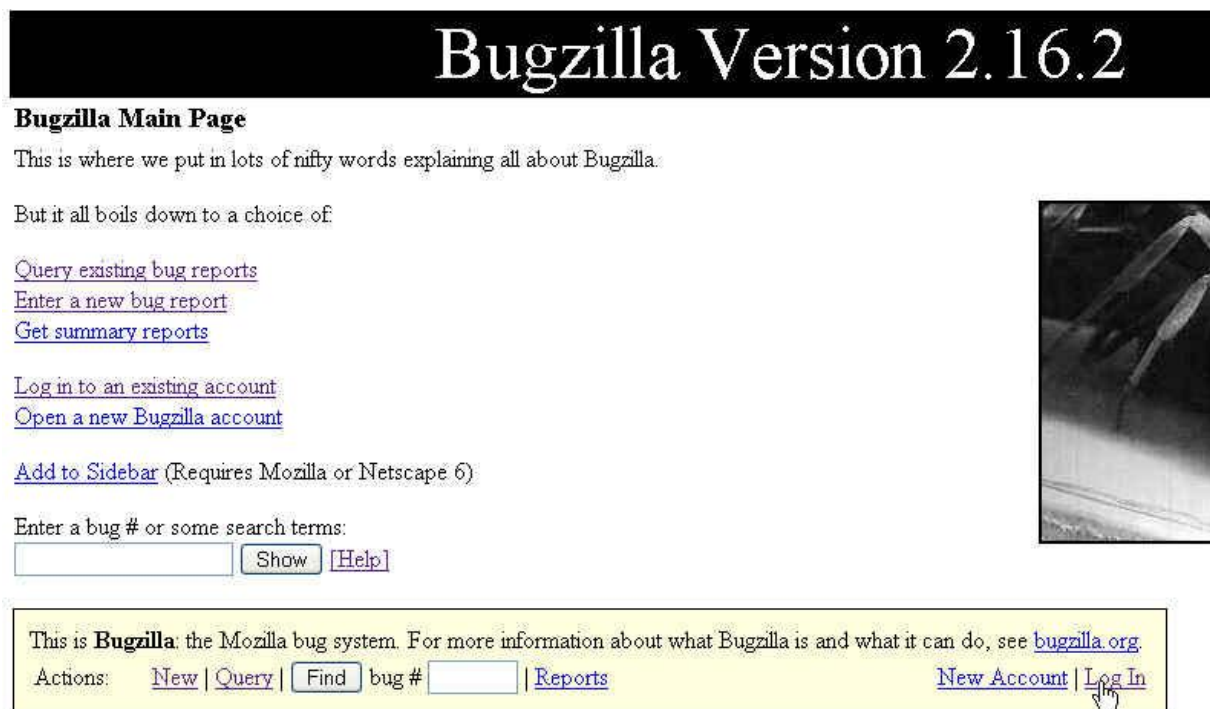
WI-0005 Dealing with an incident in WN3000.

5 Procedure

5.1 The basics, find and login to Bugzilla.

5.1.1 In order to use Bugzilla a user will need a login and a password, this they can obtain from the Test Team Leader. The username will be of the sort yourname@bugzilla.ie

5.1.2 To run Bugzilla the user needs to type the following URL in to their browser <http://bugzilla> or <http://192.168.0.54>. They will be brought to the home page which currently looks like this:



Bugzilla Version 2.16.2

Bugzilla Main Page

This is where we put in lots of nifty words explaining all about Bugzilla.

But it all boils down to a choice of:

[Query existing bug reports](#)
[Enter a new bug report](#)
[Get summary reports](#)

[Log in to an existing account](#)
[Open a new Bugzilla account](#)

[Add to Sidebar](#) (Requires Mozilla or Netscape 6)

Enter a bug # or some search terms:
 [\[Help\]](#)

This is **Bugzilla**: the Mozilla bug system. For more information about what Bugzilla is and what it can do, see bugzilla.org.

Actions: [New](#) | [Query](#) | bug # | [Reports](#) [New Account](#) | [Log In](#)

5.1.3 They must log in using the username and password as described in 4.1.1. There is a link on the homepage of bugzilla called 'Log In' this the user must click before entering in their username and password as shown below.



Bugzilla

Login

I need a legitimate e-mail address and password to continue.

E-mail address:

Password:

5.2 Receiving mails from Bugzilla

5.2.1 If the user wants to receive mails about bugs that are applicable to them then they need to set up another mail account in their email client. The settings are as follows: An email will automatically be sent every time that a new bug is entered or when the status of a bug is changed. Only those who are associated with a bug will receive an email.

POP Server: 192.168.0.54
SMTP Server: N/A
Account Name: yourname
Password: password

(N.B. For email the username is *yourname* *not* yourname@bugzilla.ie)

- Bugzilla main use: Entering New bugs
 1. Select Product
 2. Select Version & Component (If the version is not there notify the test team lead, it takes <1 min to add it)
 3. Select Priority and Severity (Normal unless a show stopper)

4. Assign the bug to the appropriate development manager unless you have been informed otherwise for this particular test pass.
 5. CC the project manager if applicable.
 6. For the summary describe concisely the bug.
 7. For the description elaborate on the steps necessary to reproduce the bug.
- An *Example bug* is shown below

Reporter: mark@bugzilla.ie

Version:

- 5.39
- 5.45
- 5.50sl
- 5.51
- 5.71

Product: UC32 Firmware

Component:

- UC32 Firmware All modules
- UC32 Keypad Firmware

Platform:

OS:

Priority:

Severity:

Assigned To: (Leave blank to assign to default component owner)

Cc:

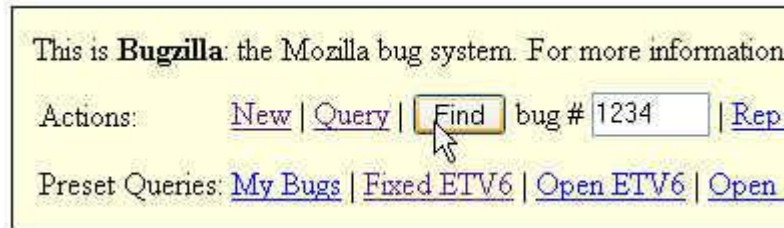
URL:

Summary:

Description:

5.2.2 Querying existing bugs.

To Query Existing bugs you can specify the bug number if known. Enter the bug number in the page footer (Yellow Box) and press Find.



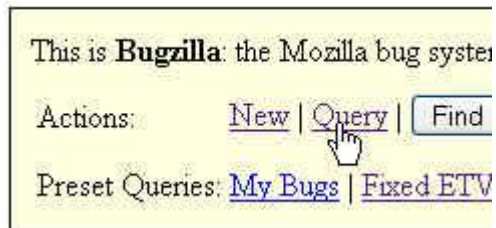
This is **Bugzilla**: the Mozilla bug system. For more information

Actions: [New](#) | [Query](#) | bug # | [Rep](#)

Preset Queries: [My Bugs](#) | [Fixed ETV6](#) | [Open ETV6](#) | [Open](#)

Or you can enter a query. To find all fixed ETV6 bugs do the following:

1. Press Query



This is **Bugzilla**: the Mozilla bug system

Actions: [New](#) | |

Preset Queries: [My Bugs](#) | [Fixed ETV](#)

2. Enter the criteria for your query, in this example all fixed ETV6 bugs

3. Select the product and the Status and the Resolution and press Search.

Search for bugs

Summary: contains all of the words/strings

Product: Component: Version:

CC1sm	All	6.0
CCView		6.6.0.35
DBViewer		6.6.0.37
ETV5		6.6.0.39
ETV6		6.6.0.53
KnownWiz		

A comment: contains all of the words/strings

The URL: contains all of the words/strings

Status:	Resolution:	Severity:	Priority:	Hardware:	OS:
UNCONFIRMED	FIXED	blocker	P1	All	All
NEW	INVALID	critical	P2	DEC	Windows 3.1
ASSIGNED	WONTFIX	major	P3	HP	Windows 95
REOPENED	LATER	normal	P4	Macintosh	Windows 98
RESOLVED	REMIND	minor	P5	PC	Windows ME
VERIFIED	DUPLICATE	trivial		SGI	Windows 2000
CLOSED	WORKSFORME	enhancement		Sun	Windows NT

NOTE: The more items that you select the more restrictive the query is, the less items the more general the query. You can deselect items by holding down CTRL and using the mouse button.

4. If you want to save the query (before you run it) and to have the option of running it in the future you can save the query in your page footer.

5. Scroll down the page and identify the 'Remember this query' section.

6. Save the query.

Remember this as my default query

Remember this query, and name it:

and put it in my page footer

Sort results by:

7. You can run this saved query by clicking on it in your footer.

This is **Bugzilla**: the Mozilla bug system. For more information about what Bugzilla is

Actions: [New](#) | [Query](#) | bug # | [Reports](#) | [My Votes](#) | [San](#)

Preset Queries: [My Bugs](#) | [Fixed ETV6](#) | [Open ETV6](#) | [Open Rest of UC32](#) | [Open](#)

5.2.3 Changing the status of a bug to fixed.

You can change the status of a bug from New or Reopened to Fixed.

1. Locate the bug in question by using the query in 4.2.2 or selecting 'my bugs' in the yellow box.
2. Enter in the comments field information about the bug that is of use in the future, information such as the version that the bug was fixed in, the consequences of the bug fix e.g. areas that were affected and why.
3. Select the Fixed radio button.
4. Press Commit.

Additional Comments:

Mention in here the implications of the code changes so that it can be recalled quickly when this bug is referenced in the future.
Mention the version that the bug was fixed in.

- Leave as **NEW**
 - Accept bug (change status to **ASSIGNED**)
 - Resolve bug, changing **resolution** to
 - Resolve bug, mark it as duplicate of bug #
 - Reassign** bug to
 - Reassign bug to owner of selected component
-

5.2.4 Reassigning a bug to a person

You can reassign a bug to an individual user for action.
Note: Assigning is different, it changes the Status to from New to Assigned. Assigning is not applicable to our lifecycle.

1. Locate the bug in question by using the query in 4.2.2 or selecting 'my bugs' in the yellow box.
2. Enter in the comments field information about the bug that is of use to the person that you are reassigning a bug to.
3. Enter in the email of the developer who is to fix the bug in the format 'developer@bugzilla.ie'.
4. Select the Reassign radio button.
5. Press Commit

Additional Comments:

Comments for a developer go in here...
Could you fix this bug as part of ECR00XX for version 6.0.1.2 etc..

Leave as NEW

Accept bug (change status to ASSIGNED)

Resolve bug, changing [resolution](#) to

Resolve bug, mark it as duplicate of bug #

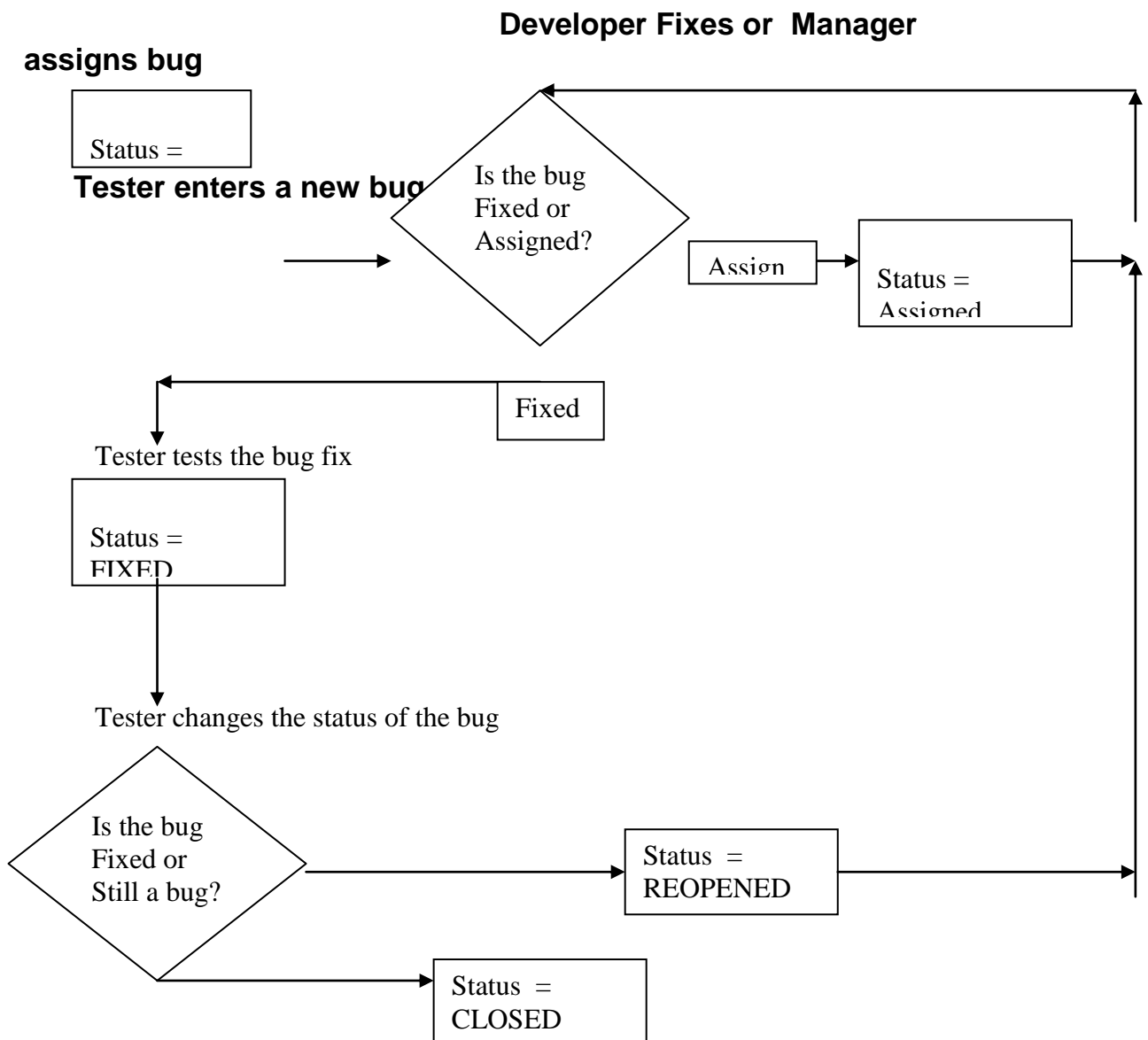
[Reassign](#) bug to

Reassign bug to owner of selected component

The Bug Status lifecycle.

1. Tester enters bug, Status = NEW
2. (Optional cycle) Development manager assigns bug to a developer, Status = ASSIGNED
3. Developer follows work instruction 005, fixes the bug and gets it signed off by the development manager who changes the status to fixed
4. Tester tests fix and either reopens bug or closes bug. Status = REOPENED (see next point) or CLOSED.
5. If the tester reopens the bug, it is then re-assigned to the relevant Development Manager.

Tester enters bug. (email sent to) Developer who fixes or assigns bug to another developer



Form 105

This form must be used to handover any build of Software to the SW Test dept.

- 6 Work carried out under ECR/CWP No.:
- 7 Application and associated files, and this form, handed over in ZIP file (insert zip file name):
R:\Test\ForTest\
- 8 Files included in this build:

		Version
Application Name		
DLL Name		
Other File(s)		

- 9 Instructions to install/set-up:
- 10 Bugs Fixed in since last build handed over to test, i.e. since build _____(give bug numbers & attached signed-off bug fix reports):
- 11 Other changes in this build:
- 12 The following tests have been performed on this build (complete unit test report should be attached):
- 13 Runlog extract for this build (or series of builds since last test handover):

- 14 The release application has been compiled from source code checked out from source safe and placed in the appropriately named subdirectory of R:\Windows Group\Program Name\Version.Build.
- 15 The release application has all the necessary files to run on a clean machine with WN3000 installed.
- 16 The release application runs on a clean machine with WN3000 installed.
- 17 The release application meets all the requirements as outlined in the requirements matrix/ECR.

18

The release application contains the correct version and build number in each of three locations, namely:



- **By right clicking the application in explorer and selecting version.**
- **On start up on the splash screen.**
- **In the help about box.**

Signed:

Developer _____ Date: _____

Test Decision:

REJECT



ACCEPT



- Assigned SWT No.:

Tester:

_____ Date: _____

Form 123

- 1 This form is related to WI-0064 SW Test Acceptance of non-released Firmware from Company x Work Programs (CWPs) & Engineering Change Requests (ECRs).
- 2 Steps to perform before the Test departments Acceptance of a completed firmware project. The steps, where appropriate, should be completed by both the developer and by the tester.

2.1 Fill in the details for the Firmware:

Controller _____

Version

Date _____

2.2 History of changes applicable to this firmware, including bug fixes, feature requests implemented:

**2.3 Comply that the firmware has been compiled from source code checked out from source
safeFirmware\Controller\ProjectName**

Developer _____

2.4 Comply that the firmware has no debugging code in it.

Developer _____

2.5 Comply that the firmware meets all the requirements as outlined in the requirement's matrix.

Developer _____

Tester _____

2.6 Comply that any changes to the spec have been updated.

Developer _____

Tester _____

4) BUG COUNTS:

Software		Count	<u>Firmware</u>		Count		
Bug Fixed	<i>As per Handover</i>		%	Bug Fixed	<i>As per Handover</i>		%
	Actual				Actual		
	Bugs Not Fixed			Bugs Not Fixed			
	New Bugs Found			New Bugs Found			
	Bugs Reactivated			Bugs Reactivated			
Total Bugs Open	<i>Blocker</i>			Total Bugs Open	<i>Blocker</i>		
	Critical			Critical			
	Major			Major			
	Normal			Normal			
	Minor			Minor			
	Trivial			Trivial			
	Enhancement			Enhancement			

5) SW Bugs:

6) FW Bugs:

9 NON-FUNCTIONAL PROBLEMS ENCOUNTERED IN PERFORMING THIS TEST

10 CONCLUSION (Inc. Pass or Fail)

11 ATTACHMENTS TO THIS DOCUMENT

Attach list of open bugs.

12 RESULTS ACCEPTANCE SIGN-OFF:

<i>Tested and accepted by:</i>	_____ <i>(Tester) Date:</i> _____
	_____ <i>(Tester) Date:</i> _____
	_____ <i>(Tester) Date:</i> _____
<i>Reviewed and accepted by:</i>	_____ <i>(Developer) Date:</i> _____
<i>Reviewed and accepted by:</i>	_____ <i>(Quality Manager) Date:</i> _____