5th International Conference
EuroSTAR '99, November 8 - 12, 1999, Barcelona, Spain

Risk Based Testing and Metrics

Risk Analysis Fundamentals and Metrics for software testing including a Financial Application case study

Ståle Amland

Hulda Garborgsv. 2,
N-4020 STAVANGER, NORWAY

Phone: +47 51 58 05 87 Mobile: +47 905 28 930 FAX: +47 51 58 55 24

E-mail: stale@amland.no

Abstract

This paper provides an overview of risk analysis fundamentals, focusing on software testing with the key objectives of reducing the cost of the project test phase and reducing future potential production costs by optimising the test process. The phases of Risk Identification, Risk Strategy, Risk Assessment, Risk Mitigation (Reduction) and Risk Prediction are discussed. Of particular interest is the use of metrics to identify the probability and the consequences of individual risks (errors) if they occur, and to monitor test progress.

The body of this paper contains a case study of the system test stage of a project to develop a very flexible retail banking application with complex test requirements. The project required a methodology that would identify functions in their system where the consequence of a fault would be most costly (either to the vendor or to the vendor's customers) and also a technique to identify those functions with the highest probability of faults.

A risk analysis was performed and the functions with the highest risk exposure, in terms of probability and cost, were identified. A risk based approach to testing was introduced, i.e. during testing resources would be focused in those areas representing the highest risk exposure. To support this approach, a well defined, but flexible, test organisation was developed.

The test process was strengthened and well-defined control procedures were introduced. The level of test documentation produced prior to test execution was kept to a minimum and as a result, more responsibility was passed to the individual performing the test. To support this approach, progress tracking metrics were essential to show the actual progress made and to calculate the resources required to complete the test activities.

1 Introduction

The risk based approach to testing is explained in six sections:

1. **Risk Analysis Fundamentals:** Chapter 2 contains a brief introduction to risk analysis in general with particular focus on using risk analysis to improve the software test process.
2. **Metrics:** Chapter 3 gives a basic introduction to the metrics recorded as part of the case study contained in this document.
3. **The Case:** Chapter 4 is the first chapter of the case study. It explains the background of how the methodology was implemented in one particular project
4. **The Challenge**: Chapters 5 and 6 further summarise what had to be done in the case project, why it should be done and how it should be done.
5. **The Risk Analysis**: Chapter 7 explains how the probability and cost of a fault was identified. Further, it discuss how the risk exposure of a given function was calculated to identify the most important functions and used as an input into the test process.
6. **The Process and Organisation**: Chapter 8 goes through the test process and discusses improvements made to the organisation and processes to support the risk based approach to testing in the case project.

In addition, chapter 9 briefly discusses the importance of automated testing as part of a risk based approach. Some areas for further research and of general interest are listed in chapter 10.

2 Risk Analysis fundamentals in software testing

This chapter provides a high level overview of risk analysis fundamentals and is only intended to be a basic introduction to the topic. Each of the activities described in this chapter are expanded upon as part of the included case study.

According to Webster's New World Dictionary, risk is "the chance of injury, damage or loss; dangerous chance; hazard".

The objective of Risk Analysis is to identify potential problems that could affect the cost or outcome of the project.

The objective of risk assessment is to take control over the potential problems before the problems control you, and remember: "prevention is always better than the cure".

The following figure shows the activities involved in risk analysis. Each activity will be further discussed below.

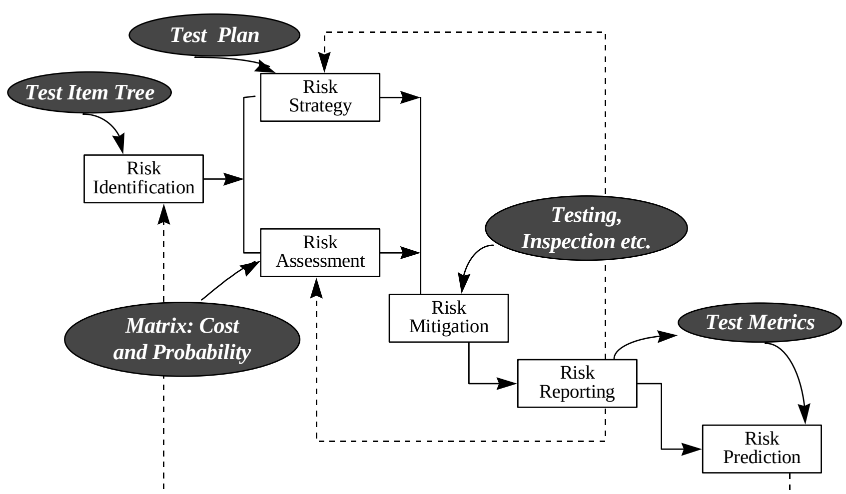


Figure 1: Risk analysis activity model. This model is taken from Karolak's book "Software Engineering Risk Management", 1996 [6] with some additions made (the oval boxes) to show how this activity model fits in with the test process.

2.1 Risk Identification

The activity of identifying risk answers these questions:

- Is there risk to this function or activity?
- How can it be classified?

Risk identification involves collecting information about the project and classifying it to determine the amount of potential risk in the test phase and in production (in the future).

The risk could be related to system complexity (i.e. embedded systems or distributed systems), new technology or methodology involved that could cause problems, limited business knowledge or poor design and code quality.

2.2 Risk Strategy

Risk based strategizing and planning involves the identification and assessment of risks and the development of contingency plans for possible alternative project activity or the mitigation of all risks. These plans are then used to direct the management of risks during the software testing activities. It is therefore possible to define an appropriate level of testing per function based on the risk assessment of the function. This approach also allows for additional testing to be defined for functions that are critical or are identified as high risk as a result of testing (due to poor design, quality, documentation, etc.).

2.3 Risk Assessment

Assessing risks means determining the effects (including costs) of potential risks. Risk assessments involves asking questions such as: Is this a risk or not? How serious is the risk? What are the consequences? What is the likelihood of this risk happening? Decisions are made based on the risk being assessed. The decision(s) may be to mitigate, manage or ignore.

The important things to identify (and quantify) are:

- What indicators can be used to predict the probability of a failure?
  The important thing is to identify what is important to the quality of this function. This may include design quality (e.g. how many change requests had to be raised), program size, complexity, programmers skills etc.
- What are the consequences if this particular function fails?
  Very often is it impossible to quantify this accurately, but the use of low-medium-high (1-2-3) may be good enough to rank the individual functions.

By combining the consequence and the probability (from risk identification above) it should now be possible to rank the individual functions of a system. The ranking could be done based on "experience" or by empirical calculations. Examples of both are shown in the case study later in this paper.

2.4 Risk Mitigation

The activity of mitigating and avoiding risks is based on information gained from the previous activities of identifying, planning, and assessing risks. Risk mitigation/avoidance activities avoid risks or minimise their impact.

The idea is to use inspection and/or focus testing on the critical functions to minimise the impact a failure in this function will have in production.

2.5 Risk Reporting

Risk reporting is based on information obtained from the previous topics (those of identifying, planning, assessing, and mitigating risks).

Risk reporting is very often done in a standard graph like the following:

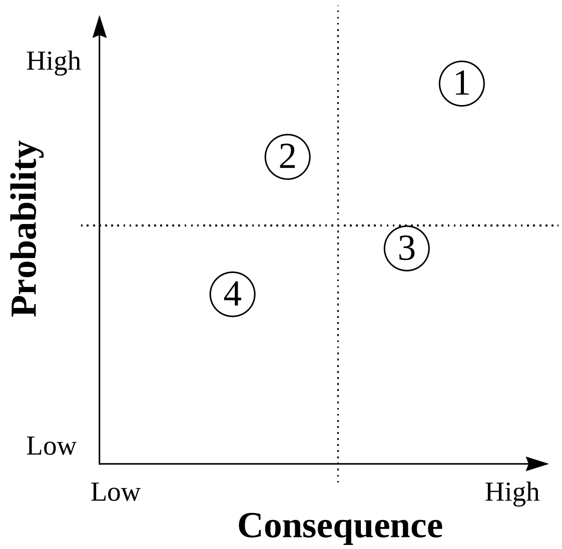


Figure 2: Standard risk reporting - concentrate on those in the upper right corner!

In the test phase it is important to monitor the number of errors found, number of errors per function, classification of errors, number of hours testing per error, number of hours in fixing per errors etc. The test metrics are discussed in detail in the case study later in this paper.

2.6 Risk Prediction

Risk prediction is derived form the previous activities of identifying, planning, assessing, mitigating, and reporting risks. Risk prediction involves forecasting risks using the history and knowledge of previously identified risks.

During test execution it is important to monitor the quality of each individual function (number of errors found), and to add additional testing or even reject the function and send it back to development if the quality is unacceptable. This is an ongoing activity throughout the test phase.

3 Metrics

This chapter will give a very brief introduction to metrics used in this document. There are several reasons to use metrics, for instance:

- Return on investment (cost / benefit analyses)
- Evaluate choices, compare alternatives, monitor improvement
- Have early warning of problems, make predictions
- Benchmark against a standard or in competition

This chapter will not give a complete picture of use of metrics. For those of you interested in reading more about metrics Norman E. Fenton & Shari Lawrence Pfleeger, 1997 [8] is recommended as a good source of information.

In this document we will make the distinction between metrics used for measuring progress and metrics used for the prediction and probability of faults.

3.1 Metrics for Progress Tracking

Metrics used for measuring progress:
1. the number of tests planned, executed and completed
2. the number of faults per function
3. the number of hours used in testing per fault found
4. the number of hours used in fixing per fault (to correct the error and return the function to re-test)

The metrics were reported graphically and trend analysis applied. For instance the information about "test planned, executed and planned" was compared with information about "faults to be fixed and actually fixed". The reason was to have an early warning of a resource problem if the number of not completed tests increased at the same time as the number of faults to be fixed were increasing.

Based on the information above, it was possible to calculate "Estimated to Complete" in number of hours, i.e. resource requirements to complete the test project. This was of course very important information in a project based on reducing risk and dynamic resource allocation to the most critical areas.

3.2 Metrics to predict probability of faults

A completely different type of metric is used to identify probability of faults in the system. Identifying indicators that were expected to be of importance per function did this. Indicators could be "Changed functionality since previous release", size of function (i.e. number of lines of code), complexity (this could be functional complexity or structural complexity), quality of design documentation etc. A number of 1, 2 or 3 (i.e. low, medium or high) was given to each indicator per function as well as a weight to handle different importance between the indicators.

Now a probability of having a fault could be calculated per function and compared to the other functions in that system. This information should then be combined with information about the consequence of a fault in each function.

Based on this information it will now be possible to "rank" the list of functions based on risk exposure (probability and cost of a fault).

4 The Case

The rest of this paper will discuss a case study using the risk based approach to software testing, relating the different activities to the activity model discussed in the previous chapter.

4.1 The Application

This paper is based on the system test stage of a project developing a retail banking application. The project included an upgrade of a Customer Information System being used by clients as a central customer, account and product database, and a complete reengineering of a Deposit Management System. The project scope included reengineering of the data model, technology change from IMS/DL1 to CICS/DB2, rewrite from JSP COBOL to COBOL-2 and a completely new physical design. During this rewrite large investments were done in productivity tools, design, quality assurance and testing.

The project started in June 1994 and was delivered in October 1995. The project total was approximately 40 man years over 17 months. This paper documents experiences from the system test stage, which consumed approximately 22% of the total project resources.

The applications consist of approximately 300 on-line transactions and 300 batch programs, a total of 730,000 SLOC[1] and 187 dB2 tables. This is the server part only, no client-GUI was tested in this project.

4.2 The Scope

The system test stage included:

1. **Technical System Test,** i.e. what is usually referred to as environment test and integration test. Due to differences between the development environment and the production environment, the system test stage had to test all programs in the production environment. During system test the test team had to do the integration test of the on-line system by testing and documenting all on-line interfaces (called modules). The team also had to perform the integration test of the batch system(s) by testing and documenting that all modules had been called and also testing the complete batch flow.
2. **Functional System Test,** i.e. black box testing of all programs and modules to detect any discrepancies between the behaviour of the system and its specifications. The integration test verified that all modules had been called, and that the functional system test was designed based on application functionality.
3. **Non-functional System Test**. The system test also tested the non-functional requirements, i.e. security, performance (volume- and stress-test), configuration (application consistency), backup and recovery procedures and documentation (system, operation and installation documentation).

As for all projects, the time and resources were limited. At the beginning of construction (programming), the system test strategy was still not agreed upon. Since the development project was a very large project to the vendor and therefore consumed nearly all available resources, the number of people with experience available for test planning was limited.

The final system test strategy for the system test was agreed approximately one month before end of construction, and the time for planning was extremely short. A traditional approach to system test planning based on test preparation done in parallel with design and construction, could therefore not be used.

The following project stages were executed before the system test[2]:

- **Project Initiation - PI** (organising the project, staffing and development environment)
- **Requirement Analysis - RA** (documents the functional requirements to the application)
- **Logical Design - LD** (data model and process model)
- **Physical Design - PD** (program design - executed as part of construction)
- **Construction and Unit Test - CUT** (programming and testing, including a 100% code coverage test)

5 The Challenge

Why did the vendor need a Risk Based Approach to the System Test?

Because:

- **The Available Calendar Time was limited**. The development project had a very short time frame. The construction and unit test stage was also delayed, so the time available for system test had to become even shorter! The applications are very flexible, and therefore very complicated to test. The calendar time did not allow for a thorough testing of all functions. Focus had to be put on those areas representing the largest risk if a fault occurred. The vendor needed a methodology to identify the most critical areas to test.
- **There were limited available resources before the end of construction!** Midway through construction a limited number of senior analysts were available for system test preparation, one for on-line and one for batch, in addition to management. The estimates to build all identified test scripts were 8 - 10 people in each of the teams (on-line and batch) until system test start! A test methodology had to be developed based on limited resources during test preparation and using most resources during test execution.
- **There were several risk factors.** Due to strict quality control of the unit test (including 100% code coverage requirements), the modules were expected to be of good quality when entering system test. However, a lot of other factors indicated trouble:

[1] SLOC = Source Line of Code, excluding comments

[2] The methodology was based on LBMS' Systems Engineering, see Systems Engineering [4].

- The project utilised a new development environment for programming, debugging and unit test based on Microfocus COBOL Workbench on PC, running CICS with OS/2. The exact effect of the learning curve on the programmer's productivity by starting with a tool all new to the organisation, was unknown.
- The development environment for on-line proved to be fairly good. However, the JCL-support for large batch streams was poor. Therefore, batch integration tests between units were not extensive and represented a risk to the integration test part of the system test.
- The system test was exclusively executed on the IBM mainframe after the source code was transferred. A test bench was developed on the IBM mainframe, executing transactions as a "dumb-terminal" client. This represented a risk since The vendor did not have any experience of the difference between the PC development environment and the mainframe environment. Differences in SQL-implementation were discovered during the project.
- Because of the vendor's expansion, a lot of the programmers were new to the company and though well educated, most of them were new to the application area and to the technology being used.
- The number of people involved was high (approximately 50 at peak) and the development time was short (planned to 17 months), this was one of the largest projects done by the vendor ever. Even though the vendor has been conducting large projects in the past, available experience from projects with this size in a compressed time frame, was limited.

What did the vendor do?

- **The System Test Strategy** document had to be rewritten. The customer did receive the preliminary version of the strategy explaining a "traditional well documented test" with everything documented prior to test execution. We had to convince the customer that this new approach was "as good as the original one, except that the new one was feasible, given the calendar time and resources available". The System Test Strategy would define the "minimum level of testing" including level of documentation for all functions and identify how a Risk Analysis would be used to identify functions to be focused on during test execution.
- We had to perform a **Risk Analysis** to identify the most critical areas both to the customer and to the vendor. A methodology had to be identified and implemented.
- **The System Test Process and Organisation** had to be improved, and even "optimised". This included defining the test process by preparing procedures, planning the test, controlling the process, progress tracking, and defining roles and responsibilities. The vendor had to convince the customer about the feasibility of the new strategy and prove the quality of the process and the product to be delivered. To document the test progress and to communicate this to the customer, became a key issue.
- **Automated Testing** was part of the contract. Initially we intended to use automated testing for all on-line functions. As the resources and time became very limited, automated testing became part of the risk based strategy, i.e. it was used to handle those on-line transactions with most faults.

The rest of this paper will document the implementation of the risk based strategy by the vendor, showing required changes to the system test process and the organisation.

6 The Strategy

The project started with a **Traditional Approach to testing**, i.e. the test should be prepared with input and output as part of the design and construction stages, prior to system test start. However, it was obvious as time passed by and only limited resources were available to prepare the System Test, that this strategy was impossible to fulfil.

The original system test strategy document (based on a traditional test approach), identified the following test structure for both on-line and batch testing:

1. **System Test Plan**, documenting the test scope, environment and deliverables, test control procedures, test tools to be used, test schedule and phases, and listing start and stop criteria related to each phase.
2. **Test Specification**, i.e. a detailed break down of the application into testable units.

3. **Test Cases**, i.e. documentation of what to test, basically listing all requirements enabling a tester to easily read them.
4. **Test Scripts**, i.e. documentation of how to test "step by step", including test data to be used by the tester.

Implementing a structure like the one above is very time consuming, especially step 4 - documenting test scripts.

Midway through construction it became obvious that it was impossible to document everything before end of construction. Either the project would be delayed, or the test planning process had to be changed.

The main problem at this stage was the preliminary system test strategy document delivered to the customer. How do you have the customer accept that you will not be able to document all tests prior to test execution as thoroughly as you originally intended to? By convincing him that the new process will improve the product quality!

The key words became "**Risk Based Approach**" to testing. We agreed with the customer (*reference to the risk activity model in chapter 2 is given in italic)*:

1. The vendor will test all functionality in the application to "a minimum level" (in addition to all interfaces, and all non-functional tests). This will not be documented prior to the test, but logging of details for all tests (i.e. input, expected output and actual output), will after test execution, prove this "minimum level of testing" (***Risk Strategy***).
2. All test cases ("what to test") will be documented prior to test start and will be available for the customer to review (***Risk Strategy***).
3. Only **highly qualified testers**, i.e. system analysts experienced in the application area, were to be utilised for testing, and the testers will be responsible for planning all "test shots", including providing test data and documenting the executed tests. (Tools were available to the tester for documenting the tests) (***Risk Strategy***).
4. The vendor will do a risk analysis together with the customer to identify those areas of highest risk, either to the customer or to the vendor (***Risk Identification and Risk Assessment***).
5. Based on the Risk Analysis, the vendor will focus "extra testing" in those areas of highest risk (***Risk Mitigation***).
6. "Extra testing" will be planned and performed by a specialist in the application area, that are not involved in the "minimum level of testing" (***Risk Mitigation and Risk Reporting***).

The 6 bullet points above cover all activities from Risk Identification through Risk Reporting. How risk reporting was used as input to risk prediction is explained later.

The customer approved the idea, and the vendor was ready to start. The project was now in a hurry and had to do the following:

1. Complete the documentation of all test cases (what to test) for on-line and batch
2. Perform the Risk Analysis for on-line and batch and plan any "extra testing".
3. Document the new risk based test process, including procedures, check lists, training the testers and preparing the test organisation.

7 The Risk Analysis

(Risk Identification, Risk Strategy and Risk Assessment)

The risk analysis was performed prior to system test start, but was continuously updated during test execution. Separate analysis was performed for on-line and batch.

The vendor developed a model for calculating the risk exposure based on:

- the probability of an error (in the on-line transaction or batch program), and

- the cost (consequence) of an error in the corresponding function, both to the vendor and the customer (in production).

Similar methodologies have been documented by others, see Øvstedal and Stålhane, 1992 [5]. However, this paper explains a practical implementation of the methodology.

There are three main sources to the Risk Analysis:

1. **Quality of the function** (area) to be tested, i.e. quality of a program or a module. This was used as an indication of the probability of a fault - P(f)[3]. The assumption is that a function suffering from poor design, inexperienced programmer, complex functionality etc. is more exposed to faults than functions based on better design quality, more experienced programmer etc.
2. **The consequences of a fault in the function as seen by the customer** in a production situation, i.e. probability of a legal threat, loosing market place, not fulfilling government regulations etc. because of faults. This consequence represents a cost to the customer - C(c).
3. **The consequences of a fault in the function as seen by the vendor**, i.e. probability of negative publicity, high software maintenance cost etc. because of a function with faults. This consequence represents a cost to the vendor - C(v).

The assumption was that the cost to the customer is equally important in the risk analysis to the cost of the vendor, and the calculated Risk Exposure of a function Re(f) would then be:

$$\mathrm{Re}(f) = P(f) * \frac{C(c) + C(v)}{2}$$

An example of areas with different risk profiles is **area of interest calculation** and **area of printing internal reports**. A fault in interest calculation could easily end up with a legal threat for the bank using the software, while a fault in the printing of internal reports not had to be known by the public at all.

7.1 Risk Analysis - On-line

At the time the on-line risk analysis was performed very little information was still available from the construction teams. The process was very simple, however not compromising the software quality and satisfying to the customer.

On-line Risk Analysis steps:

1. **Prior to System Test:** The vendor asked the customer to set up a "Top-20" list of on-line transactions which they viewed as the most critical transactions. All transactions on that list would go through extra testing.
2. **Throughout System Test:** The vendor would add to the list of critical transactions based on the number of faults found, those transactions were also subjects for extra testing. The number of faults could vary due to specification and design quality, programmers lack of knowledge in the application area, lack of knowledge about the development / production environment etc.

Extra Testing consisted of two elements:

1. **Additional testing by product specialist**. A separate checklist was developed, and the exit criteria for the transaction to pass were at least one hour of continuos testing without any errors detected. All transactions on the customer's Top-20 list and all transactions with totally more than 10 faults ("the bad ones") found in the system test by the vendor, would go into this process. The total of 10 faults could include faults

[3] According to B. Beizer: **fault** - incorrect program or data object; a bug, **error** - incorrect behavior resulting from a fault; a symptom. See Beizer, 1990 [2].

detected because of environment problems, configuration problems in the build process, functional faults and errors in the documentation.

2. **Regression Testing Executed in an Automated Test Tool.** All transactions with more than 4 functional errors would go through full regression testing after last bug-fix was proved to be correct. The regression test would replay all tests executed by the tester during the test execution process, using the test tool AutoTester, see AutoTester [1].

7.2 Risk Analysis - Batch

The risk analysis of the batch areas was done when construction was midway through. The project had now gained substantial information about the application area and the test and production environment. The quality in the design stages had also been proved by implementation.

The application's batch areas were split into sub-areas (or functions) as part of the logical design stage. Those functions were now used in the risk analysis.

A sample list of batch functions is

- Interest Calculation
- Penalty Calculation
- Profitability Analysis
- Delete of Data
- Reporting
- Capitalisation

According to the formula:

$$\mathrm{Re}(f) = P(f) * \frac{C(c) + C(v)}{2}$$

the challenge was to identify the cost of a fault and to identify indicators of quality, i.e. what is the probability of a fault in this function?

We used a very simple, though satisfactory method to develop these two elements, i.e. cost of a fault and probability of a fault, for all batch functions. The vendor invited the project management, product specialists, designers, programmers, people working with application maintenance and test management, into a meeting. This team had to agree on the cost of a fault and the probability of a fault for each function.

1. **The cost of a fault** was indicated by a number from 1 to 3 where 1 represented minimum cost in case of a fault in this function. Elements to be considered were:
   - Maintenance resource to allocate if a fault occurred during customer's production (given the vendor would provide 24-hours service).
   - Legal consequences by not fulfilling government requirements.
   - Consequences of a "bad reputation".
2. **The Probability of a fault** was indicated by giving 4 indicators a number varying from 1 to 3, where 1 was "good", i.e. the probability of a fault was low. The indicators were:
   - **Changed or New Functionality**. The project was a re-engineering of an existing application and the change of functionality varied from function to function. The programs had to be rewritten anyway, but if the functionality was not changed, at least there was an exact specification of the function.
   - **Design Quality** would vary, depending on the function (some functions were all new), and by design and application experience of the designer. This was measured by counting number of Change Requests to the design (for further explanation of the project process, see section "The Process").
   - **Size.** We assumed that the number of sub-functions within a function would affect the number of faults introduced by the programmer.
   - **Complexity** (logical)**.** The programmer's ability to understand the function he was programming will usually effect the number of faults.

For the cost related to each batch function we used the average of the Customer's Cost C(c), and the Vendor's Cost C(v).

The indicators used to calculate the Probability of a fault for a particular function P(f), were weighted, i.e. the weight would vary from 1 to 5, rating 5 as the most important indicator of a function with poor quality.

The weights used by The vendor were:

1. Changed or New Functionality - 5
2. Design Quality - 5
3. Size - 1
4. Complexity - 3

An example of calculated risk exposure for the batch function "Close Account" is shown in the figure below. The Risk Exposure Re(f) was calculated for all batch functions and the list was sorted to identify those areas to be focused during testing. The Probability P(f) is calculated as the Weighted Average of a particular function divided by the highest Weighted Average of all functions, giving the probability in the range [0,1].

	Cost			**Probability**						
Func.	C(v)	C(c)	Avrg. C	New Func. 5	Design Quality 5	Size 1	Compl. 3	Weight Avrg.	Proba-bility P(f)	Risk Exp. funct. Re(f)
Close Accnt.	1	3	2	2	2	2	3	7,75	0,74	1,48

Figure 3: Example of calculated Risk Exposure for the batch function "Close Account". The vendor's cost of a fault is low (1), but the customer's cost is supposed to be high (3). The average cost is then 2 (Avrg. C). The probability is calculated by calculating the weighted average (Weight Avrg.) divided by the highest Weighted Average for all functions (which in this example was 7,00), giving a probability in the range [0,1]. The Risk Exposure Re(f) is the multiplication of the Average Cost and the Probability.

8 The Process

(Risk Strategy, Risk Mitigation, Risk Reporting and Risk Prediction)

The limited time and resources available made it very important to have a well defined test process. This included:

- Interfaces to design, construction and unit test stages
- Deliverables with defined quality standards
- Test preparation and execution procedures stating entry and exit criteria for each test phase
- Control procedures to handle scope changes and issues
- Well defined organisation and responsibility including training of the testers
- Progress Tracking

8.1 Interface to Design and Construction and Unit Test

The overall test process was based on Logical Design (LD) and Physical Design (PD) giving the base for the build of the Test cases. Those test cases had to adhere to defined quality standards. The test cases were given to the tester together with the Test Execution Procedure. The tester would prepare test data and report any problems during test execution. The problems would be documented in a Problem Tracking Document and would be passed on to the test team leader for verification before going to the fix team leader.

We called it a “problem” document because it could be any kind of problem, including program faults, but also environmental problems, test data problems, change requests etc.

After the problem was fixed the problem tracking document was passed back to the test team for re-test. If the test passed, the function would be evaluated as “Good” or “Bad”, based on number of errors discovered in this function. If this function did include a high number of faults, then extra testing would be applied, either by building a complete set of regression test scripts for the test tool AutoTester[4] or by additional manual testing. Finally, the quality of the function would be evaluated as “good”, and the test result would go through extensive quality control (QC), i.e. checking completeness and accuracy and finally verifying all formalities in the quality assurance process (QA).

A Fix might also include a Change Request (CR) to either the Logical or Physical Design. The test team leader would typically originate the CR for Logical Design whilst the fix team leader would typically originate the CR for Physical Design.

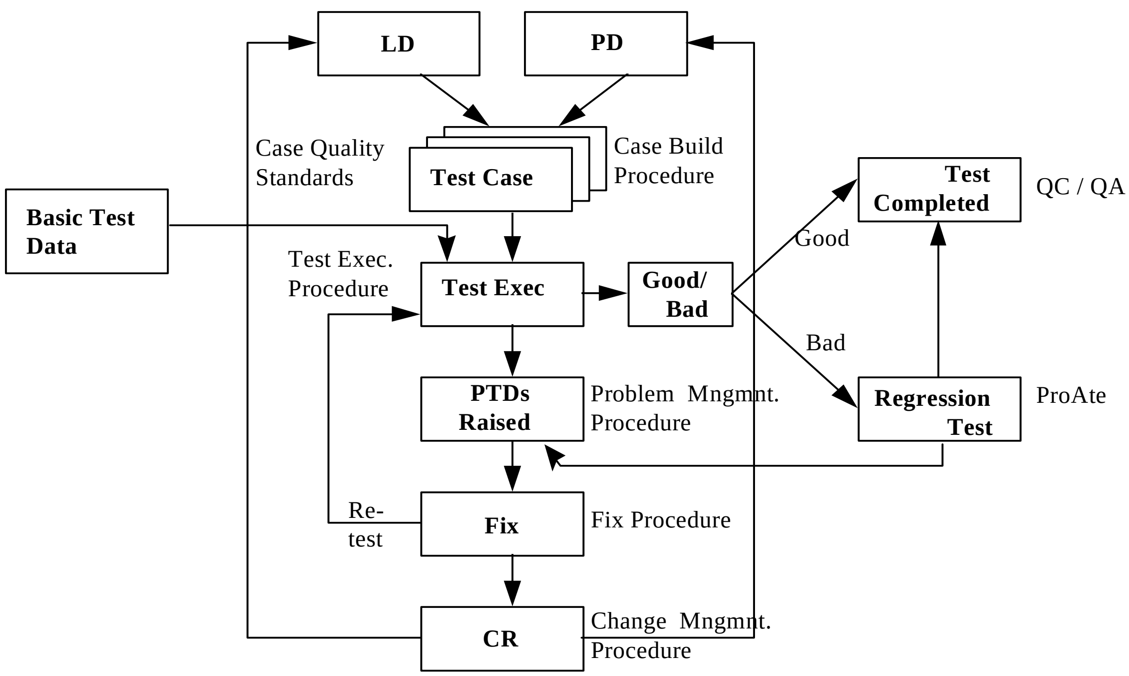


Figure 4: This figure shows the work flow of the System Test Process, from design documents to test case, from test case to test execution and then a problem might occur. When the problem is fixed and re-tested it might be accepted by the QA/QC procedures. After the testing is completed, the function might go through regression testing, depending on number of faults detected. For on-line the regression testing is performed in ProAte, i.e. The vendor's AutoTester Environment (see footnote 4).

The risk based approach was based on the assumption that the tester himself prepared the test script including test data and that he documented the test result. This activity was important to the success of the risk based approach since the approach puts more responsibility on the tester than a traditional approach.

8.2 Documentation Structure and Deliverables

Prior to test execution the following documents were prepared:

[4] The program AutoTester from AutoTester Inc., see AutoTester [1].

1. **The System Test Plan**, including scope, roles and responsibilities, deliverables with level of detail, procedures overview and test schedule.
2. **The System Test Specification**, giving a system break down for testing, i.e. identifying all functions being tested.
3. **The System Test Cases**, for each function in the test specification the test case would identify **what** functionality to test by listing each validation rule to verify, all combinations of input data to verify etc. The test case would typically state "Test a future date in the Posting Date field", not including the actual date to be used in the test.

During the on-line tests the tester executed test shots and documented them in test scripts with input data used, expected result, and test result. A test tool provided the tester with logging capability to document the input data being used, logging the expected result and logging the actual result. The tools forced the tester to log the expected result prior to logging input data and actual result. This made the testers prepare their test shots carefully. Again, this methodology of testing and documenting and the tool were important for the risk based approach success, i.e. the methodology forced the tester to be prepared and the tool supported in the documentation process.

The idea was to use the same structure for batch testing, in which we did not succeed. Due to the fact that a batch run takes more time to execute and therefore the number of "retries" is limited, the batch test scripts including test data and expected result, had to be completed before the processing of a batch cycle.

The test structure is visualised in the figure below.

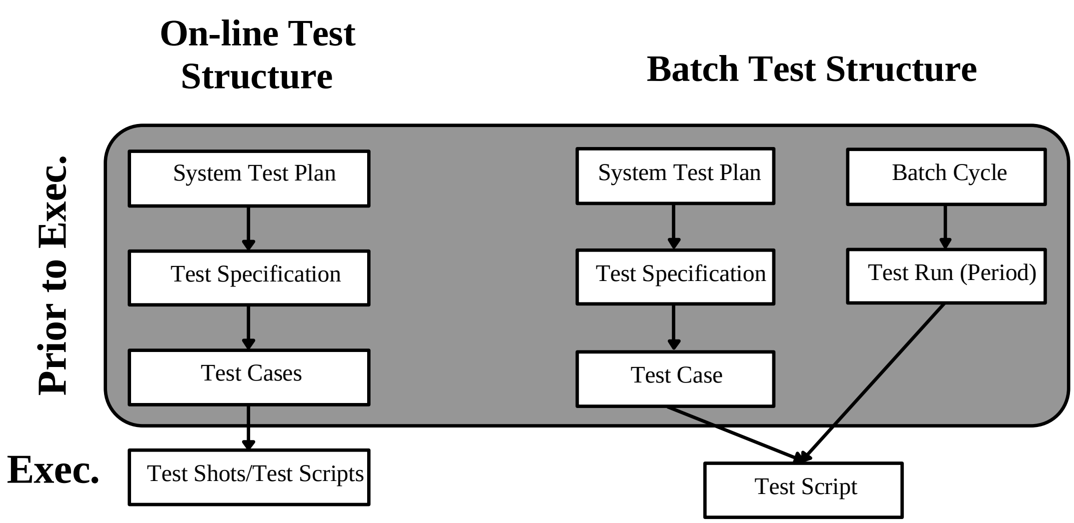


Figure 5: System Test Documentation Structure, on-line and batch, as implemented according to the risk based approach. The plan, specification and test cases were developed prior to test execution but the test scripts / shots were developed and documented during test execution. In the batch test the test scripts were more complete prior to test execution than in on-line. Batch preparation included also design of test runs (single processing days) and batch cycles (a series of test runs, e.g. daily run, month end and year end).

This test structure allowed for a "delayed" resources profile by utilising highly qualified testers during test execution and a limited number of qualified persons in test preparation. This was absolutely necessary since there were only limited resources available until end of construction.

8.3 The Organisation and the Test Planning

The resource profiles below show accumulated resource profiles for:

1. Estimated resource requirements for a traditional approach (**Original Estimate**)
2. Estimated resource requirements for a risk based approach (**Risk Based Estimate**)

3. Actual resource usage, based on the selected Risk Based Approach (**Actual**)

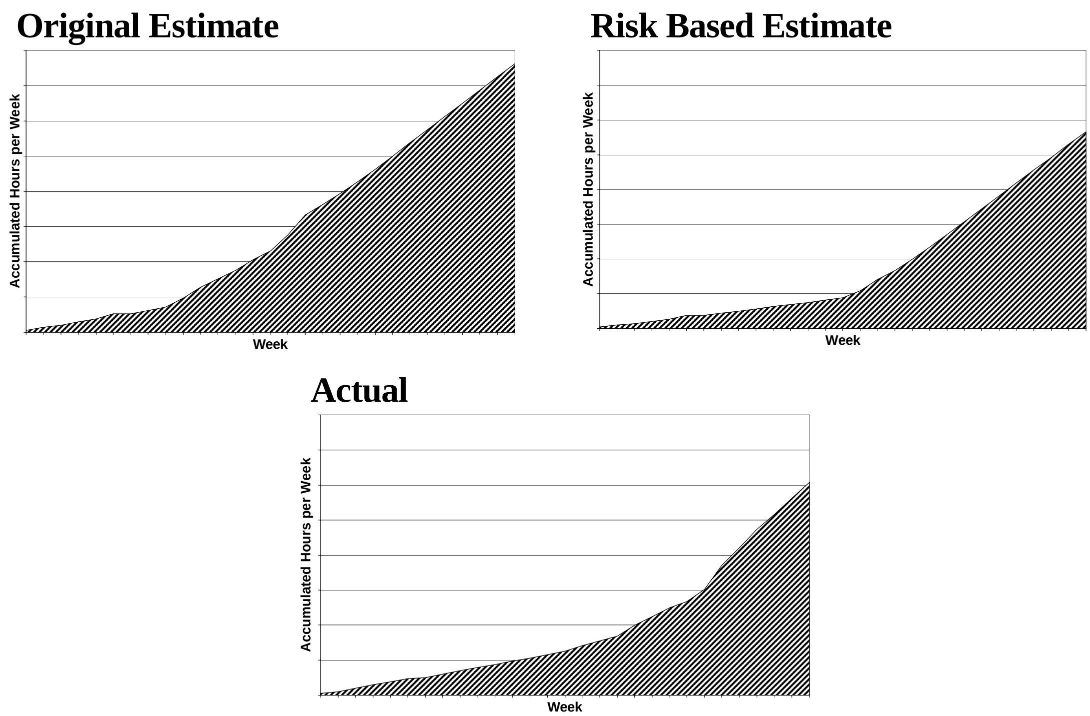


Figure 6: Resource profiles for Original Estimate (i.e. Traditional Approach), Risk Based Estimate (Risk Based Approach) and Actual (i.e. actual accumulated number of hours spent).

The graphs show that the risk based approach consumed less resources relative to the original estimate based on a traditional test approach.

The risk based test approach is highly dependent on using qualified testers, i.e. testers with experience within the application area and preferable with experience within the test environment. The reason is that the tester himself will build the actual test scripts during test execution, including test data. It is obvious that inexperienced testers will need a lot of training to be productive with this approach. However, training must be executed to make even the qualified testers familiar with the new test approach. The vendor executed several pilot projects, both in the on-line and the batch area to verify the test methodology, the accuracy of procedures and to train the testers. This was done through the short test planning phase.

Another criteria for success for the risk based approach was an efficient, dynamic and flexible test organisation. Over time, it proved to be essential that “the testers (i.e. test team leader) do prioritise”. Whenever there were discussions about which faults to correct first, the decision had to be based on the risk analysis and what functions needed to be tested most, not which fault was most convenient for the programmers to fix.

The organisation shown below proved to be efficient at supporting the Risk Based Approach.

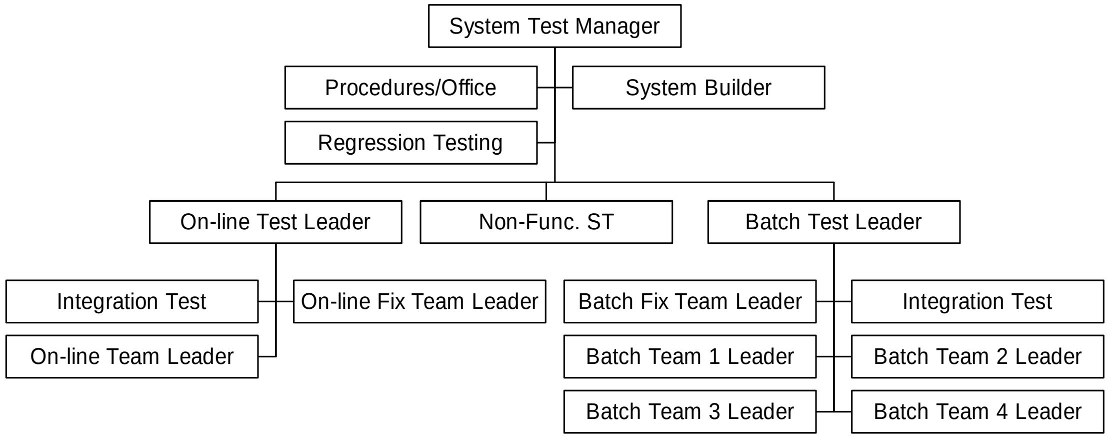


Figure 7: The Risk Based Approach to testing requires a flexible organisation, focused on fixing bugs related to critical functions.

What made this organisation efficient was the combination of a fixed high level structure combined with flexibility on the "detailed" level.

The fixed high level structure is represented by:

- **Centralised services.** The follow-up on procedures and production of test documentation was taken care of by the project office ("Procedures / Office" in the chart). Similarly, the configuration control was taken care of by one role - the System Builder, and the automated regression testing was executed by one "Regression Testing" team.
- **Centralised Progress tracking.** Progress tracking was only done by the system test manager, although all individuals recorded their own progress every day.
- **Separate teams for on-line, batch and non-functional testing.** The documentation requirements and the test execution procedures were quite different for all three teams, and the organisation had benefits from not training the testers in more than one area.
- **Well defined responsibility.** The team leaders were responsible for the preparation of all test cases prior to test execution and also for reviewing the result after the test. In addition there was a quality control function to assure completeness and accuracy, and finally a quality assurance of the formalities.

The flexibility on the detailed level is represented by:

- **Shared resource pool for testers and fixers.** This is not represented in the graph above, but was very flexible. E.g. instead of the test team waiting for fixes to be implemented, they would participate in the fix process, and vice versa. This would also improve the testers knowledge about the system, and finally improve the testing quality.
- **Shared team management,** i.e. the on-line team leader would be the same person as the on-line test leader, and the batch team leader would be the same as the batch test team leader. This is done to assure that the focus is on testing (not on fixing), and the prioritisation of which modules to fix is done based on the risk analysis and not on which modules are "easiest to fix".

8.4 The Control Procedures

The project implemented separate procedures for control issues (i.e. changes to scope, process or schedule not related to system design) and change requests (i.e. changes related to system design and implementation). All projects will be affected by issues and change requests, but the risk based approach made this project even more vulnerable to changes. The reason is that the planning process had been limited and the detail test scripts were not prepared until test execution. Usually several faults are identified during the test planning phase, but by the risk based approach to testing, the planning phase is part of the test execution.

All change requests had to be accepted by the product manager outside the system test team.

8.5 Progress Tracking and Progress Indicators

(Risk Reporting and Risk Prediction)

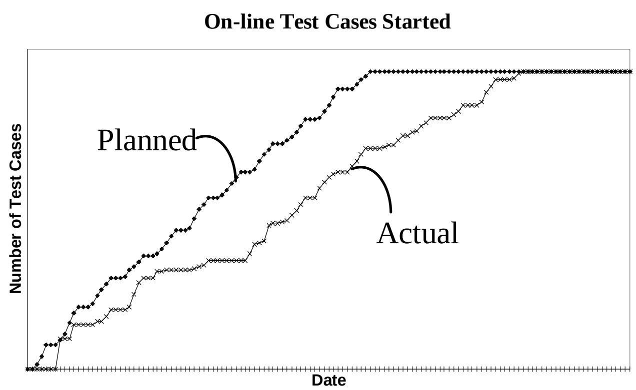
The introduction of the risk based approach made it very important to document the test progress to the customer as well as the quality of the test. The quality of the test was documented as part of the output documentation from the test execution, including the complete listing of the test log from the test tool.

The progress tracking was critical to have the customer believe in the product and to believe in the end date. The reporting included basically two elements:

1. The number of tests started and completed according to plan.
2. Indicators to show the load of faults found and corrected.

8.5.1 On-line Progress Tracking

The following graphs show the on-line tests started and completed. Because of the limited material prepared prior to test execution, the quality control of the test documentation prior to execution was very limited. This made the quality control (QC) and quality assurance (QA) processes during test execution even more critical. Therefore, the curve most interesting to the customer was "tests actually completed from QA" in the graph On-line Test Cases Completed.



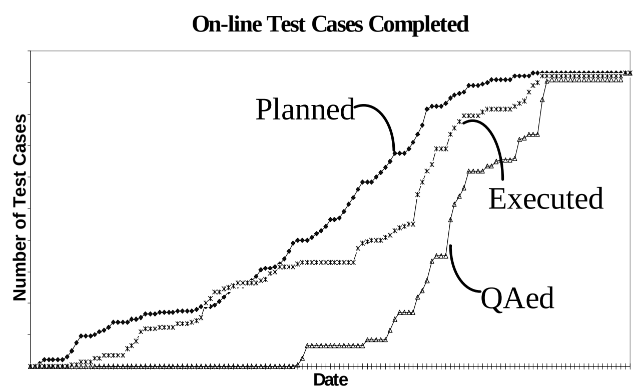


Figure 8: Progress Tracking. To the left is a graph showing planned and actual of test cases started. The graph to the right visualises test cases completed, showing planned complete, actually completed by tester (executed) and actually completed from QA.

8.5.2 Batch Progress Tracking

The batch process of ProDeposits is very complex. Approximately 300 batch programs constitute a daily batch run. A traditional approach to batch testing for The vendor would have been to set up one batch system and process day by day, fixing problems as they occurred.

The approach was to run as many test runs as possible as early as possible, to identify problem areas (i.e. areas of high risk) and to focus the test in those areas.

The consequence was that 3 sub-systems were set up, each with a batch cycle consisting of 12-15 batch runs, i.e. processing 12-15 periods. A period (point of time) is a single day, a week end, a month end, a year end etc. Each batch cycle would be processed at least 3 times for all three systems during the system test period.

If possible, the complete batch cycle was completed, not waiting for fixes of faults identified in one batch run before continuing. The result was an early detection of problem areas and the possibility of focusing the test.

Because of the planned strategy to run all batch cycles three times per system, the "number of tests started" did not make any sense in batch testing as it did in on-line. The result was that the plan for batch testing was calculated as the total number of batch tests (i.e. number of verifies[5] to be executed), evenly spread over the total number of days for batch testing. As a result of this, the progress did not look good in the beginning when a lot of outstanding integration tests were executed. However, after a while the progress improved, and during the last few weeks the graphs showed a steady slope.

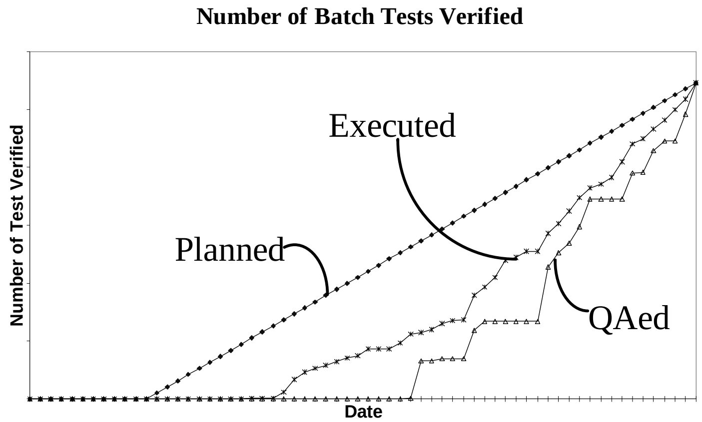


Figure 9: Progress Tracking Batch. The plan was calculated based on number of verifies to be executed and number of days of testing. The graph also shows number of verifies actually executed and number of tests QAed by date.

The graph above gave the customer a snap shot of the current situation. In addition we needed some indicators that provided the customer with good visibility of, and therefore confidence in, the test process.

8.5.3 Progress Indicators

We used two indicators, one related to the test process and one related to the fix process. The first one showed number of faults reported to the fix team and number of faults fixed (i.e. reported back to the test team). The second indicator showed number of faults reported back to the test team for re-test from the fixers and number of faults re-tested. The second indicator also included a graph showing number of fixes from the fix team being rejected by the testers as part of the re-test.

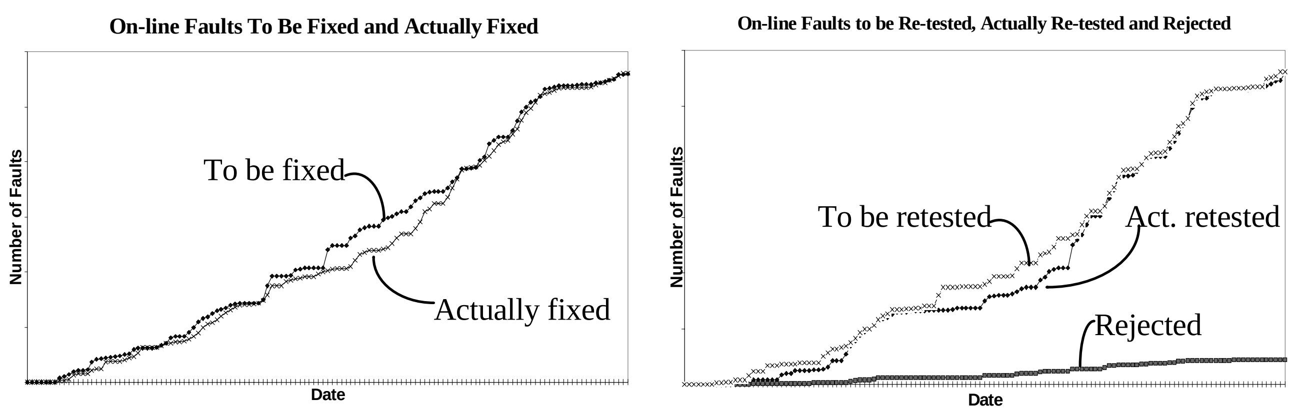


Figure 10: Progress Indicators. The left graph shows the number of faults delivered to the fix team and number of faults fixed. The graph at the right shows the number of reported faults that have been fixed and returned to re-test (to be re-tested) and the number of faults actually re-tested. The lower curve is the number of fixes being rejected in re-test.

[5] A verify document is a document with the expected result of a particular batch test. The document will list all results to look for after a test run. The number of expected results can vary depending of type of test. To “execute a verify” is to compare the actual result with the expected result after a test run.

Similar graphs to the above were developed for batch faults.

8.5.4 Estimated To Complete (ETC)

The calculation of ETC for test projects is always complex, and even more complicated when the preparation work is as limited as in this project. Again, the need for indicators to predict the number of resources needed to meet the end date, was essential to the test approach chosen.

We closely monitored the number of hours spent in testing and in fixing related to the number of faults identified and fixed. The following graphs were used for both, on-line and batch.

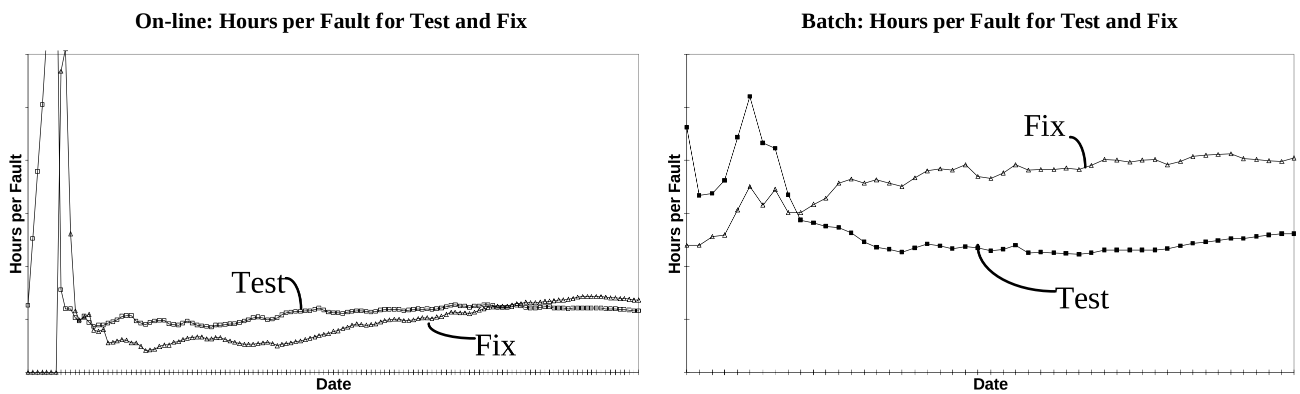


Figure 11: Estimated to Complete. The number of hours testing per fault found and number of hours analysis / programming per fault fixed were used as indicators to calculate ETC.

In addition to the number of hours per fault the following numbers were used for calculating the ETC for on-line:

1. Number of faults found per on-line transaction (i.e. per on-line test case)
2. Number of fixes being rejected (i.e. generating a new fault to be sent to fixing and re-test)
3. Number of remaining on-line test cases

By combining 1, 2 and 3 above, the remaining number of faults could be estimated, and by using the numbers from figure above, the total resource requirements could be estimated.
For batch the calculation method was somewhat different. In addition to the number of hours per fault found, the numbers used were:

1. Number of faults found per verify document.
2. Number of fixes being rejected (i.e. generating a new fault to be sent to fixing and re-test)
3. Number of verify documents being accepted out of total number of verify documents reviewed
4. Number of verify documents still to be verified.

The result graphs for on-line and batch are shown in the following figure. The rising curve is the accumulated hours spent and the falling is the calculated ETC over time. It took some weeks before the ETC-calculations were reliable, but they proved to be very accurate during the last few weeks. If more historical data could have gone in to the ETC calculation, a reliable result could have been provided at an earlier stage.

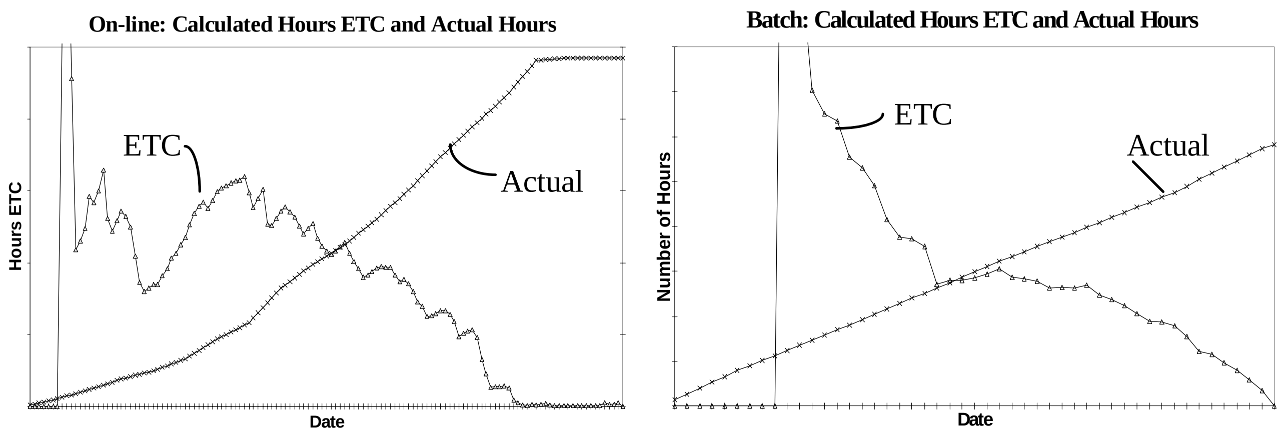


Figure 12: Calculated ETC and Actual hours spent for on-line and batch.

9 Automated Testing

(Risk Strategy and Risk Mitigation)

The project was committed to use automated regression testing by utilising the tool AutoTester from AutoTester Inc [1]. This proved to be a commitment very hard to fulfill. Originally the intention was to develop all AutoTester test scripts prior to test execution.

Due to the changed test approach, the information required to develop AutoTester scripts was not available, i.e. the test data and the scripts would be provided by the tester during test execution.

The Risk Based approach was based on each tester using AutoTester to log all test shots and to record test scripts for automated regression testing. This proved to be very complicated because:

1. The tester had to think of regression testing all through test execution. This included to plan the test data, the sequence of transactions etc.
2. All testers shared the same database. They could easily damage each others test data if not paying attention.

The project used 25% of the total test resources for on-line, in automated regression testing. The regression test team managed to regression test 15% of all on-line transactions, and found 2.5% of all faults.

The recommendation for the next similar project will be:

1. Let the manual testers focus on doing manual tests, using a tool for documentation / recording without thinking automation. The result should be readable, not re-playable. The tester should be able to set up his own test data within his limits.
2. Set up a separate database for automated regression testing.
3. Select the “worst” transactions for automated regression testing.
4. Identify a separate test team to focus on automated testing.
5. Do not start recording for automated regression testing until the function is stable, i.e. most faults have been identified and fixed.
6. Over time develop a “lifetime” test script for all transactions, i.e. an automated test script to be used as an installation test at customer’s site.

For those wanting to start with or improve automated testing, I will strongly recommend a book being published this summer (1999) by Dorothy Graham and Mark Fewster "Automating Software Testing" [7].

10 Further Research

As a “test” the McCabe complexity, see McCabe, 1976 [3], was checked for a random list of the 15 on-line transactions with the highest number of faults identified and 15 on-line transactions with the lowest number of faults identified. The result showed that the McCabe complexity in average is 100% higher for those with a high number of faults than for those with a low number of faults.

This material however, needs more investigation. Particularly interesting is the analysis of the function’s logical design to be able to identify functions with a potential of a large number of faults, at an early stage.

11 Acknowledgement

A lot of people have helped in making this document. AVENIR ASA, Norway has funded the work, and I thank Per Bakseter for being my sponsor. I will also thank Gro Bjerknes, Hans Schaefer, Bjørnar Evenshaug, Stephen Løken, Stein Onshus and John Curtin for providing valuable comments to the initial version of this paper. Special thanks to Bo Kähler, Ged Hawkins, Travers Sampson and Joy Hanson who gave me examples and corrections to the latest version of this document.

12 References

No. Reference

1. AutoTester, AutoTester Inc., 6688 North Central Expressway, Suite 600, Dallas, Texas 75206, Tel: + 1 800 328 1196
2. Boris Beizer, “Software Testing Techniques” Second Edition, Van Nostrand Reinhold, 1990.
3. McCabe, Initial paper on cyclomatic complexity definition, McCabe, T.J. 1976, A Complexity Measure, IEEE Trans. On SW Eng., Vol2, No. 4, Dec. 1976
4. Systems Engineering**,** LBMS Europe (now part of Platinum Technology Inc.), 1815 South Meyers Road, Oakbrook Terace, IL 60181, USA).
5. Øvstedal, E. Ø. and Stålhane, Tor “A goal oriented approach to software testing”, Reliability Engineering and System Safety. © 1992 Elsevier Science Publishers Ltd., England
6. Dale Walter Karolak, “Software Engineering Risk Management”, IEEE Computer Society Press, 1996.
7. Dorothy Graham and Mark Fewster, "Automating Software Testing", 1999
8. Norman E. Fenton & Shari Lawrence Pfleeger, "Software Metrics, a rigorous & practical approach", 2nd edition, International Thomson Computer Press, 1997.