PROCESS VISION
DOWN TO EARTH IT PROJECT IMPROVEMENTS

The basics of Software Quality Control

A guideline
for reviewing and testing
a software system

Willem van den Biggelaar

PROCESS VISION
KONINGSKRUIS 1
5081 XM HILVARENBEEK
TEL: 0031-(0)13 5053526

WEB: WWW.PROCESSVISION.NL
EMAIL: INFO@PROCESSVISION.NL

Foreword

This paper gives the basics of software quality control. First it will explain what software quality control is.
It will give in a nutshell what reviewing is and how to use it. It will explain the V-model, the several test phases and the deliverables of the test phases. Test methods are discussed and test tooling is briefly touched. If possible, metrics are given on the several subjects.

The light bulbs in this document indicate useful practical tips.

Definitions & acronyms

Audit	An independent examination of a work product or a set of work products to assess compliance with processes, specifications, standards, contractual agreements or other criteria.
Bug	A defect
Code coverage	The percentage of the code that has been activated after the execution of the software application.
Debugging	Part of development where the programmer of the code checks the code after bugs (by means of testing or usage) are found by himself or somebody else.
Defect	An instance in which a requirement is not satisfied [Fagan].
Driver	Small test program feeding the unit with test data and reporting test results.
End-customer	The person who has paid for the system being developed and will be the owner and user of it.
Failure	Malfunction caused by a defect or bug.
Functional requirements	All "hard" requirements of a system. The basis functions of the system. See also "Non functional requirements".
KLOC	Kilo Lines Of Code: 1000 lines of code
Major Defect	A defect of which the inspector says it is a major
Minor Defect	All non major defects
Module	Smallest part of a software system according the V-model.
Non functional requirements	All "soft" or so-called *quality requirements* of a system. E.g. usability, maintainability, compatibility, performance. See also "Functional requirements".
Quality Assurance	Activities that measure the *process,* identify defects and suggest improvements. The direct result of these activities is changes to the process.
Quality Control	Activities that measure the *product,* identify defects and suggest improvements. The direct result of these activities is changes to the product. The output of quality control activities is often input to quality assurance activities
Review	An evaluation of a document aiming to find defects.
Scenario (OO term)	A scenario is an instance of a use case, and represents a single path through the use case. Thus, one may construct a scenario for the main flow through the use case, and other scenarios for each possible variation of flow through the use case (e.g., triggered by options, error conditions, security breaches, etc.). Scenarios may be depicted using sequence diagrams.
SMART	Specific, Measurable, Accepted, Realistic and Timely
Stub	Dummy routine, simulating lower level software or hardware
Target system	The system for which the software under test is meant. For instance the actual X-ray system.
Testing	The process of finding defects in relation to a set of predetermined criteria or specifications. The purpose is not to prove that a system works, but to prove that a system does not work.

Use Case (OO term)	A goal-oriented set of interactions between external actors and the system under consideration. An actor may be a class of users, roles users can play, or other systems. A use case is initiated by a user with a particular goal in mind, and completes successfully when that goal is satisfied. It describes the sequence of interactions between actors and the system necessary to deliver the service that satisfies the goal. The system is treated as a "black box", and the interactions with system, including system responses, are as perceived from outside the system.

References

[Hatton]	Safer C, Developing software for high integrity and safety critical systems, Les Hatton, 1999
[Harry]	Harry, Mikel. "Six Sigma: The Breakthrough Management Strategy Revolutionizing the World's Top Corporations." New York, N.Y. Random House Publishers, 2000.
[Boehm]	Improving Software Productivity, B. Boehm, IEEE Computer, Vol. 20, NO. 9, 1987
[Gilb]	Software Inspection, Tom Gilb & Dorothy Graham, Addison-Wesley, ISBN 0-201-63181-4
[Fagan]	Design and Code inspections to reduce errors in program development M. Fagan, IBM System Journal volume 15 no 3
[Trew]	Demystifying Design for Testability, Tim Trew, Philips Digital Systems Lab, Redhill
[Cornett]	Code coverage analysis, Steve Cornett, Bullseye testing technology

Table of Contents

1. Introduction

1.1 What is Quality Control?

In this course we are going to talk about Software Quality Control. To find out what that is, let's do some history first.

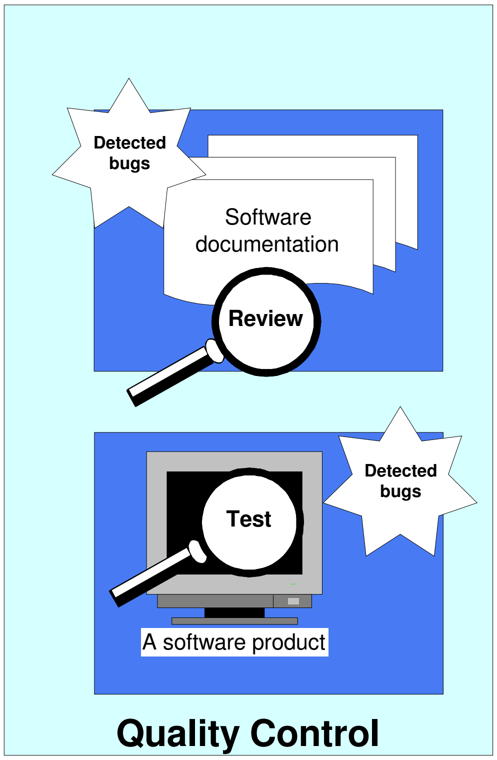
1.1.1 Debugging

Until the early 1970 most software development organizations had not clearly differentiated between testing and debugging. Life was simple, a system was built on the end-customer's wishes. Next, the end-customer used it and the programmer removed the found bugs on the fly.

1.1.2 Testing

After this chaotic period, one started to build more complex systems. The need arose to write down the end-customer wishes in the form of so-called requirements. The system had to obey these requirements and separate testing activities were born: an activity to find the faults in the system that do not correspond to the requirements. But still there was a big problem: testing was always done at the end of the cycle: it was *finding and correcting* bugs afterwards.

1.1.3 Quality Control



Nowadays, systems are that complex that they can exist of millions line of code. Real time systems have become huge applications. They are put together multi-disciplinary: mechanics-, hardware- and software engineering join forces.

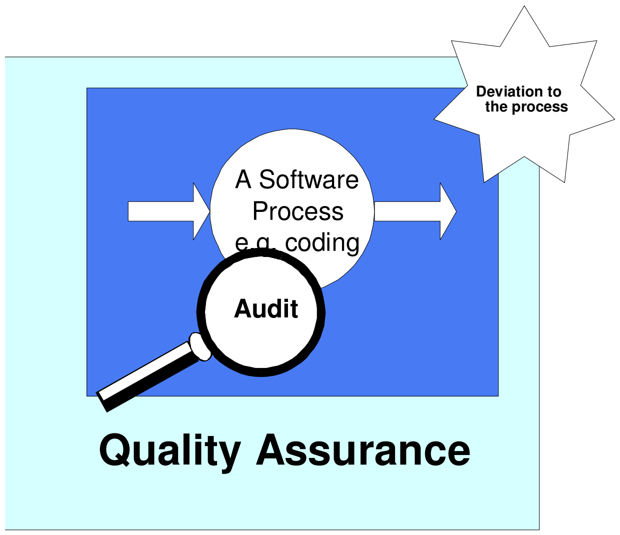
And where does it all start? At the customers requirements! This document is the first object that needs to be tested.

Testing a document is called 'reviewing': several people (all stakeholders of the document) read this document and give comment (detect bugs).

After the requirements, the software specifications and designs are reviewed. Even the code is reviewed.

Finally, the compiled and linked code is tested. Testing occurs with the focus on one or more levels (module, unit, sub-system, system) dependent on the complexity and size of the application.

1.2 What is Quality Assurance?



With quality assurance, the agreed way of working is checked against the actual way of working. For instance, if the project has planned and agreed that every new piece of code is subjected to a code review, the quality assurance officer checks if this indeed happens. The checking if quality control is performed as planned is also part of his job. A helpful instrument for the officer is an audit. By means of interviews with project members he can get a clear picture of what is going on in the project.

So, the difference between quality control and quality assurance is that the first checks the *product*, and the second checks the *process*.

1.3 What is Design for testability?

Design for testability is the measures the designer of a system takes to increase the testability of the system. Designing is out of scope of this document but a good article about this subject is [Trew].

1.4 Why Quality Control

Barry Boehm has done some research on this back in 1987 (ref. [Boehm]). He examined a number of representative software projects and looked at the cost of rework in relation to the found defects. In other words: what does a bug cost the project (or the company). Below his results are given.

If a defect in a requirement document was found in the detailed design phase, the costs would have been 70 times more than when it was found during the requirements document review. As soon as the product is released (aftercare phase), the costs are 250 times more.

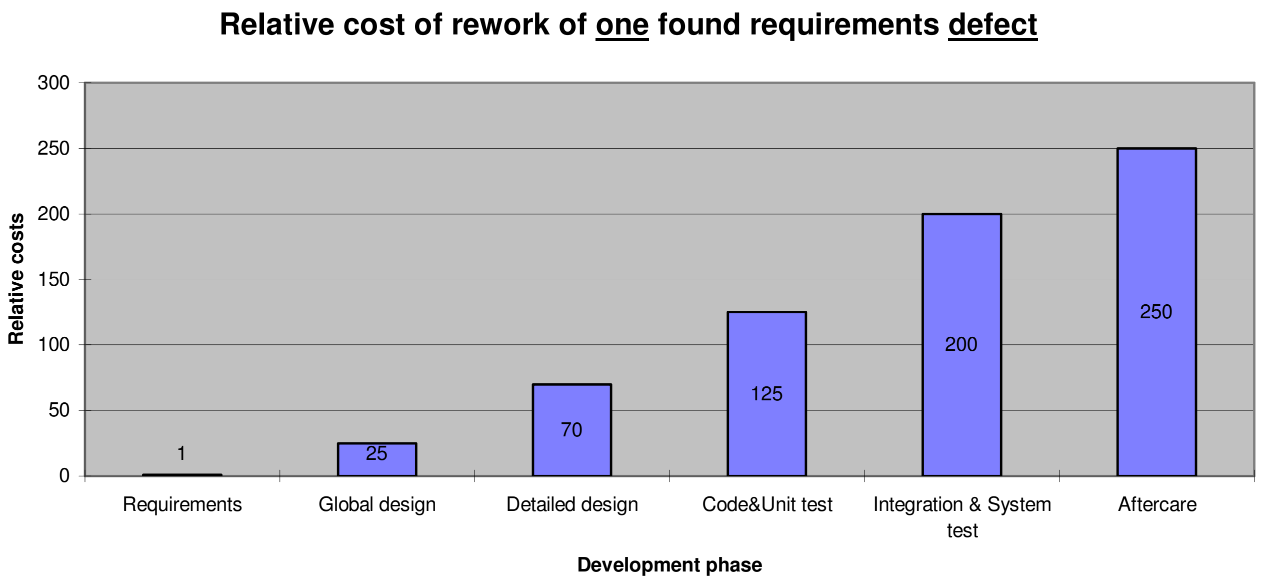
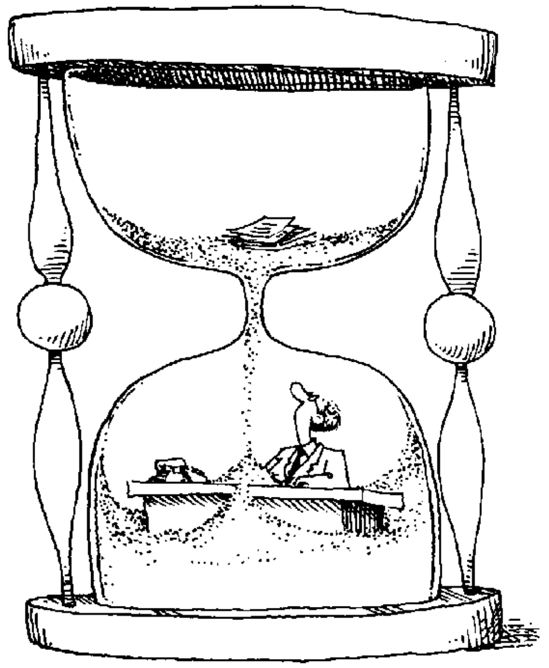


Figure 1 Relative cost of rework

A real life example:

> During development of a digital audio player marketing forgot to specify the digital output at the back of the chassis. The defect was found during production of the player: in the factory they wanted to assemble the connector for the output and found that there was no electronic connection. It took 1 more month to get this fixed. 20 Men had to work the whole month to fix this problem.
>
> What was the origin of the problem here? The factory people were not invited for the review of the requirements. If they were invited, they would certainly have triggered on this fault because a chassis of this type of player has by default such an output.

1.5 When to stop testing?



Too little testing is a sin, too much instant death

1.5.1 Stop on zero fault software?

Is testing finished when all the bugs in the system are found? How many bugs are there in a product? The answer to that question is unknown because it is impossible to find them all.

[Hatton] says:

- Safety systems (aero-planes, nuclear plants, medical applications) have an average of 1 error / 1000 lines of code (KLOC) after release.
- A reasonable commercial system has about 3 to 6 errors / KLOC after release.
- A poor system has more than 15 errors / KLOC after release.

However, [Harry] is more optimistic:

- Domestic (U.S.) airline flight fatality rates run at better than six sigma, which could be interpreted as fewer than 3.4 fatalities per million passengers - that is, fewer than 0.00034 fatalities per 100 passengers.
- The current average industry runs at four sigma (6210 defects per million opportunities).
- Internal Revenue Service phone-in tax advice, runs at roughly two sigma (308,537 errors per million opportunities). Depending on the exact definition of defect, this could be interpreted as 30 out of 100 phone calls resulting in erroneous tax advice.

History shows a lot of software failures:

1.5.1.1 Mariner 1 Venus probe loses its way: 1962

A probe launched from Cape Canaveral was set to go to Venus. After takeoff, the unmanned rocket carrying the probe went off course, and NASA had to blow up the rocket to avoid endangering lives on earth. NASA later attributed the error to a faulty line of Fortran code. The report stated, "Somehow a hyphen ("-") had been dropped from the guidance (besturings) program loaded aboard the computer, allowing the flawed (foute) signals to command the rocket to veer (koers veranderen) left and nose down...Suffice it to say, the first U.S. attempt at interplanetary flight failed for want of a hyphen." The vehicle cost more than $80 million, prompting Arthur C. Clarke to refer to the mission as "the most expensive hyphen in history."

1.5.1.2 Radiation machine kills four people: 1985 to 1987

Faulty software in a Therac-25 radiation-treatment machine made by Atomic Energy of Canada Limited (AECL) resulted in several cancer patients receiving lethal overdoses of radiation. Four patients died. When their families sued, all the cases were settled out of court. A later investigation by independent scientists Nancy Leveson and Clark Turner found that accidents occurred even after AECL thought it had fixed particular bugs. "A lesson to be learned from the Therac-25 story is that focusing on particular software bugs is not the way to make a safe system," they wrote in their report. "The basic mistakes here involved poor software-engineering practices and building a machine that relies on the software for safe operation."

For more historical failures, see appendix A.

1.5.2 Stop if run out of resources?

Developing products is always balancing between 3 major elements: time, money and quality (including functionality).

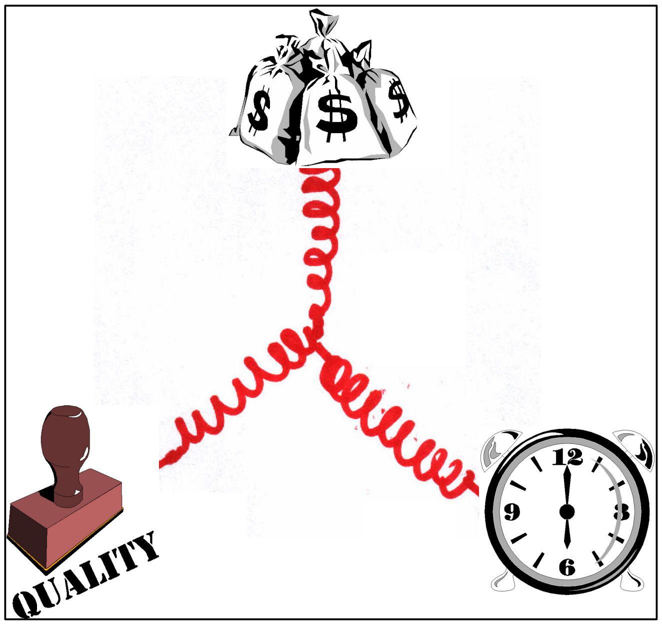


Figure 2 Balance between quality, budget and time

Many times, there is no balance:

- "We must meet our deadline. Time is running short on us. Can we not skip the module testing?"

- “We have no time left to review these documents, just authorize them and start with the coding. “
- “Is there still budget left to continue testing?”
- “Oh, did we forget the code reviews? Damn, but we had planned it, so that’s good, isn’t it?”

In many projects, the main reason to stop with quality control is that the money or the time has run out.

1.5.3 Stop on SMART criteria?

Instead of stopping at the impossible (zero fault software) or stopping for the wrong reasons (running out of resources), there is a better way: define criteria at the start of the project. These criteria have to be SMART:

Specific	not vague
Measurable	quantifiable
Acceptable	agreed upon by end-customer and project
Realistic	nothing like “the program should contain zero faults”
Timely	a timeline must be incorporated

Below examples are given of SMART criteria.

1.5.3.1 Number of found defects dropped below limit

Example: “the system has passed the system test if the number of defects the test team finds drops below 1 defect per 50 testing hours”.

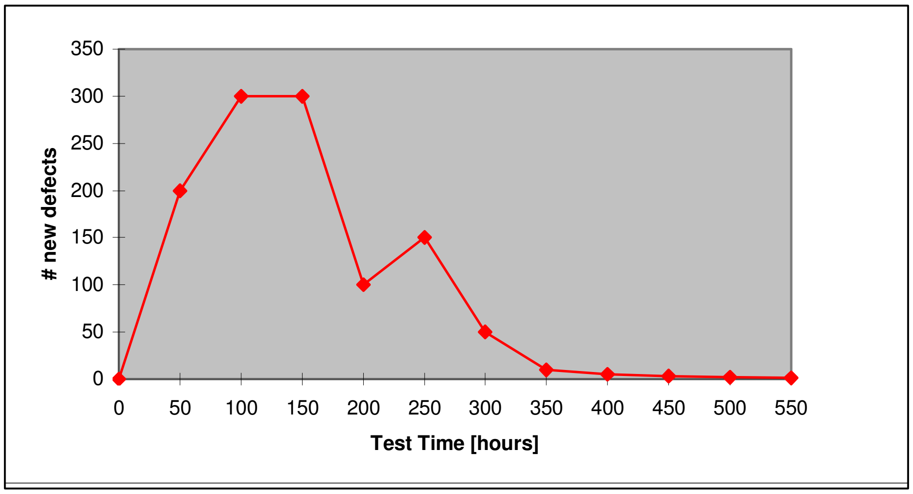


Figure 3 Number of found defects in relation to test time

Testing hours are the effective hours the system is being tested by the test team by running the tests. Not included are for example

- Hours test software is being installed
- Startup problems on the target system being tackled.

1.5.3.2 Percentage of successfully passed test cases reached target

Example: “The system is accepted if 95% of all acceptance test cases have passed successfully. From the other 5%, only minor problem reports may be open. “Successfully“ means that the logged output from all the tests within the test cases matches the expected output”.

Be aware: 100% system test case coverage does not mean the system is perfectly tested on system level. It only means that all test cases have run well. Maybe certain requirements were forgotten to be covered in test cases! That’s why the next stopping criterion is also important.

1.5.3.3 Requirement coverage reached target.

Example: “the system is accepted if all “must do” requirements are covered by test cases. 80% of the “should do” requirements are also covered.

1.5.3.4 Code coverage reached target.

There are several definitions of code coverage. Some examples:

- Statement coverage: each statement has to be executed at least once
- Function coverage: each function has to be executed at least once
- Path coverage: each possible path has to be executed at least once

The article of [Cornett] gives more definitions and examples.

It is never feasible to try to reach 100% path coverage. An example:

```
Do (20 times)
Begin
        if (a > b)
                func_A(a)
        else if (a = b)
                func_B(b)
        else
                func_C(c)
End
```

The above piece of simple pseudo-code already has 3^{20} (= 3486784401) possible paths.

Examples of feasible targets are

- “The condition/decision code coverage of all user interface modules must have reached at least 80%. For the other 20%, valid reasons are given for not covering”.
- The condition/decision code coverage of all low-level hardware interface modules must have reached at least 75%. For the other 25%, valid reasons are given for not covering”.

How is a feasible code coverage target calculated?

1. Decide which definition to use.
2. Run a code coverage tool that can give the defined coverage (see appendix B for a list of commercial tools)
3. Analyze the code and classify them:
    a. Code that is covered
    b. Code that is not covered but must be covered

   c. Code that is not covered but is not coverable without major effort in this test phase (test code becomes more than application code).
   d. Code that is not covered but must be removed (obsolete[1] code)
4. With the above classification a target is set for this module

Next the test cases for classification "b"" are adapted and the obsolete (classification "d") code is removed.

You have to define together with your customer what the criteria are when to stop testing. Document them in your test plan.

[1] Applications that exist for several years can contain already a lot of obsolete code!

1.6 When to stop reviewing?

When do you stop reviewing a document? When all bugs are found? The same answer stands here as for testing: not all bugs will be found.

The question here is: is the document mature enough for the stakeholders? Has for instance the designer enough information to start with his design if he has read the input requirements? Can the testers translate the input requirements into test cases? Sees the end-customer all his user requirements covered in the system requirements? If such questions are answered positive, the document has passed the review.

2. REVIEWING

Now let's dive into the first section of quality control: reviewing.

2.1 Definition

A lot of terms are used in companies: inspections, peer reviews, management reviews, technical reviews, brainstorm and walkthroughs. What one company means by a walkthrough is in another company an inspection. So bear that in mind when you join a company. Of course literature tells exactly the differences between all these terms but that will not help in real life.

The definition present here is:

"Reviewing is an evaluation of a document[1] aiming to find defects"

2.2 Goals

2.2.1 Main goals

- Detect a **maximum** of defects in the **earliest** possible stage of the development. Defects become progressively more expensive to eliminate if they are deleted in a later stage of development (efficiency).
- **Find** defects not correct them. Correction is the responsibility of the author.
- Have a stable product to be placed under Configuration Management
- Aim at finding **major** defects; avoid the 90% minor syndrome (spending 90% of review time finding minors). But what is a major defect? Again, as with the definition of reviews, each projects has its own definitions:
  - Anything with potentially large downstream cost consequences.
  - Anything that does not comply with a functional or non-functional requirement that will be visible to the end-user.
  - Anything that will damage the commercial value of the product.
  - Anything that changes external interfaces

  The definition presented here is:

  "A major defect is major because the inspector says it is a major"

2.2.2 Positive side effects

- Transmit knowledge among reviewers and among reviewers and author
- Enhance overall quality next to locating defects such as
  - Usability
  - Maintainability
  - Testability
- Control and enforce adherence to relevant standards
- Create public support for the product

[1] In this respect, evaluating software code is also a review: the hardcopy of the code is then the document

2.3 The roles

A person involved in a review process always has two roles:

1. A process role: moderator, recorder, inspector, commentator, author
2. A product role: principal, consumer, specialist

For instance, an architect that reviews a design can be appointed explicitly moderator (=process role) for that review and is implicitly a principal (=product role) for that design.

2.3.1 Process roles

Below the review roles in the review team are given:

Moderator	Person responsible for the process of the review. Checks compliance of the review to the standard review process. Decides if review can start. Chairman of the review meeting. Timekeeper of the meeting. Decides on outcome of meeting. Tracks rework down to closure. Checks the rework. The moderator and author can never be the same person. The moderator must be well-trained in the review process.
Recorder	Scribe of the review meeting
Inspector	Reviewer who must attends the meeting and must give comment.
Verifier	Reviewer who checks if all defects are properly solved.
Commentator	Reviewer who does not attend the meeting and may give comment
Author	Person who delivers the product to be reviewed. The author can never be the moderator.

2.3.2 Product roles

A design document needs other reviewers than a requirements document. In general, a document under review has the following stakeholders:

- Principal(s): person who wrote the higher-level document, e.g. for a module design specification, the module requirement specification is its higher-level document. The principal should check if the product under review has performed the correct translation
- Consumer(s): person who needs the product under review as input e.g. for a module design specification, the module code programmer is a customer. The consumer should check if the product under review is clear, detailed and consistent enough for him.
- Specialist(s): person who has specific (technical) knowledge needed for the document under review, e.g. user interface specialist, a concurrent programmer specialist. The specialist checks the correctness on his field

Below some examples of product roles are given:

Document Type	Reviewers		
	Principals	Consumer	Specialists
Acceptance Test Spec	User Requirements author	System Test Spec author	--
Software Unit Requirement Spec	Software System Designer Functional Requirement Spec Author	Unit interface users Unit Design authors Unit Test Spec authors Integration Manager	--
Project Plan	Department Manager	Project Team	Quality Officer
Code	Software Module Design author	Module Interface users	Colleague, C++ Coach

Table 1 "Examples of Product review roles"

2.4 The Process

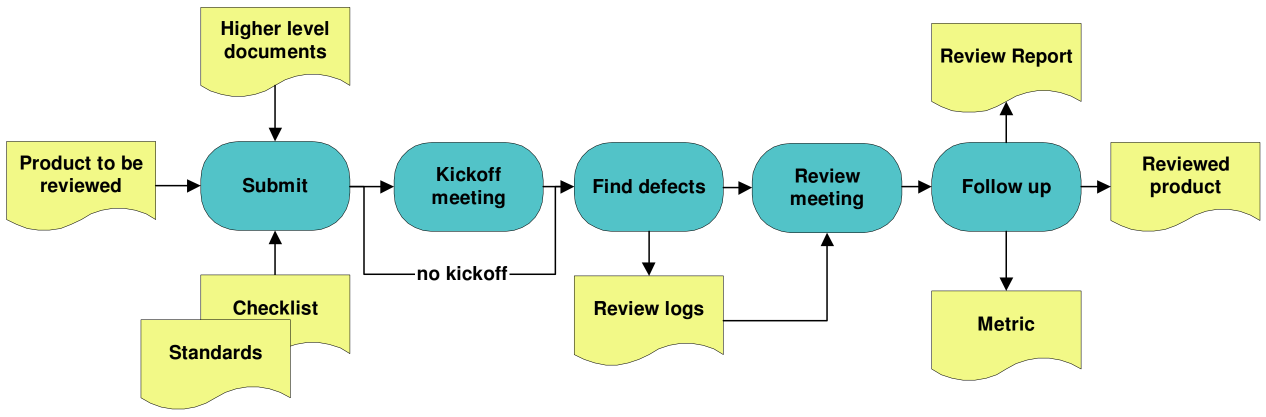


Figure 4 the review process

2.4.1 Submit

The author together with his team- or project leader defines the review team according the process roles and product roles (see previous section).

Next the moderator checks if the product is ready for review:

- 'Maturity' of the document: does it not contain too many open issues.
- Are the process and product roles filled by the right people
- If an organization uses static code checkers (e.g. QA-C++, Code Wizard, Cantata++, Lint), the output of these tools should be used as input for a code review. The moderator can then check if the code has passed these tools.

2.4.2 (Optional) Kick-off meeting

Next the moderator organizes a kick-off meeting. This meeting can be skipped if the inspectors already know the product and know what is expected of them for the review. If the meeting is skipped, the moderator takes care all documents are sent to the inspectors.

In the meeting, the author hands out the line numbered product under review and gives a technical briefing of the product. The moderator explains the process and hands outs the checklists, higher documents and standards. The inspectors leave the meeting with fully understanding their roles.

2.4.3 Find defects

The inspectors study the product to be reviewed and note their comment on the review log. Optional, the inspectors send their logs back to the moderator a few hours before the start of the review meeting. The moderator merges all defects into one defect log (sorted on page/line). He prints out the merged defect log for all reviewers and brings it with him to the meeting.

2.4.4 Review meeting

The moderator checks if the inspectors have properly prepared the review:

- Are the review logs filled in correctly
- Has the preparation time been enough? It is hard to give figures for preparation time because it highly depends on
  - Maturity of an inspector
  - Maturity of the organization
  - Type of document

  Figure 5 Checking rate optimum gives a metric on how to come to an indication for the checking rate.

The above check can be done before the meeting, if the moderator has a merged defect log.

The moderator now goes through the document page by page. The inspectors read their defects and ask for clarifications if some parts of the document are not understood. No discussion on possible solutions takes place. The author reacts on the questions and asks for clarification if a defect is not understood.

The inspectors decide if a defect is accepted or rejected. There are however companies that let the author decide: he is end responsible of the quality of the document. The problem with the latter is, that an author is most of the times under a lot of time pressure in a project and tends to reject a defect if that costs a lot of update effort.

At the end of the document, the moderator asks the inspectors if a new review is needed (maybe the document contains too many defects). If not, the review itself is accepted otherwise it is rejected and a new review must be planned.

The recorder writes down the following metric on the review report:

- Time spent by everyone on every review step (preparation, kick-off, find defects, review meeting)
- Number of accepted major defects
- Number of accepted minor defects
- Number of reviewed pages

- Outcome of the review: accept or reject

2.4.5 Follow up

The author now updates the document according the review logs. The verifier checks the rework and sends it back to the moderator. The moderator finishes the review report by signing it and completing the metric:

- Time spent by author and moderator on follow-up step

Finally the author communicates to the project that the document is reviewed.

Next the product must be authorized (releasing the product) and archived.

Finally, the review report together with the review logs are archived as proof that the product has been officially reviewed.

2.5 Guidelines

- Don't confuse poor products and poor reviews (process went wrong)
- No discussions about solutions in the meeting
- Avoid using 'negative' words in the meeting
- Judge the product, not the author
- No more than 7 people in a review meeting
- Don't start with unprepared people

2.6 Hints & tips

2.6.1 Pre-review documents

Before an official review, it is wise to let colleagues check your document when it is in draft state (especially for new documents or if you are new in the field of expertise).

2.6.2 Define viewpoints

To get certain focus and to save time, viewpoints are defined. Each inspector gets a special task on which he has to check. Examples of viewpoint are

- Check if product is according standards
- Check on consistency with all high level documents
- Check on multi-threading aspects
- Check on usability of defined screens

Define these viewpoints before the kick-off meeting.

2.6.3 Collect review logs before review meeting

The review team must hand in the review logs one day before the actual review meeting takes place. The moderator then merges all defects into one sheet and sorts them on page/line number. At the review meeting he hands out this merged list of defects.

Advantages:

- Moderator knows a day before the meeting if everybody has actually prepared the review properly. If not, he can cancel the meeting on beforehand instead of at the start of the meeting.
- The moderator guides the meeting more easily because he knows exactly who has found a defect on which page.

- The recorder has now a very small role. He only writes done accept/reject of the defects, generated actions and the metric data on the review report.

Disadvantages:

- For many people, writing defects on a review log already is a big step, writing it electronically is a huge step.

2.6.4 Sample large documents

If a document to be reviewed is 50 pages or more, the method of [Gilb] can be used. Select for instance 5 pages that together give an impression of the quality of the document. Review those pages first. If no more than 0.2 majors and 2 minors per page are found, trust that the other pages have the same quality and accept the document. If more defects are found, ask the author first to solve the defects found and let him update also the other 45 pages. Next re-review 5 other pages.

Drawback here is how to find those first 5 pages.

2.6.5 Log typos defects on hard copy of document

You can log typos (misspelling) on the document itself instead of using review log forms because it's a more natural way of noting during reading.

Be aware that these scribbles must be readable to the author. Do not allow putting majors also in the document, majors must be put on the review log form.

Another drawback: a nice checkpoint for the moderator to see if the inspectors have actually put time into the document is the output on the review log forms.

I have worked in a company where reviewing was not taken very seriously: inspectors came into the meeting, claiming they had put 2 hours of effort in preparation. During the meeting, you saw them reading the document as it had been for the first time. They scribbled the defects on the document during the review. One of the measures we took in that organization was that every defect had to be logged on a log form and that inspectors who did not have a filled in log form was expelled from the meeting. It worked, even if it seems a very childish measure.

2.6.6 Only discuss majors during the meeting

If a moderator sees at the beginning of the meeting that a lot of majors are found (by simply asking the inspectors how many majors they have found), he can decide to only discuss the majors. By default the minors are then accepted. If the author has a problem with a minor, he can discuss this off-line with the inspector. The advantage here is that the meeting will be shorter and there will be a focus on the real problems.

2.6.7 Verify majors only

When there are many minors to verify, introduce the following rule:

- Always verify majors
- Verify minors by sampling
- Never verify typo's

2.7 Metric

The most essential metric are:

- Total time spent on inspecting the document by everybody per review
- Total number of accepted defects found per review
- Total number of inspected pages per review

With the above information, the optimum checking-rate for the organization is calculated. An example follows:

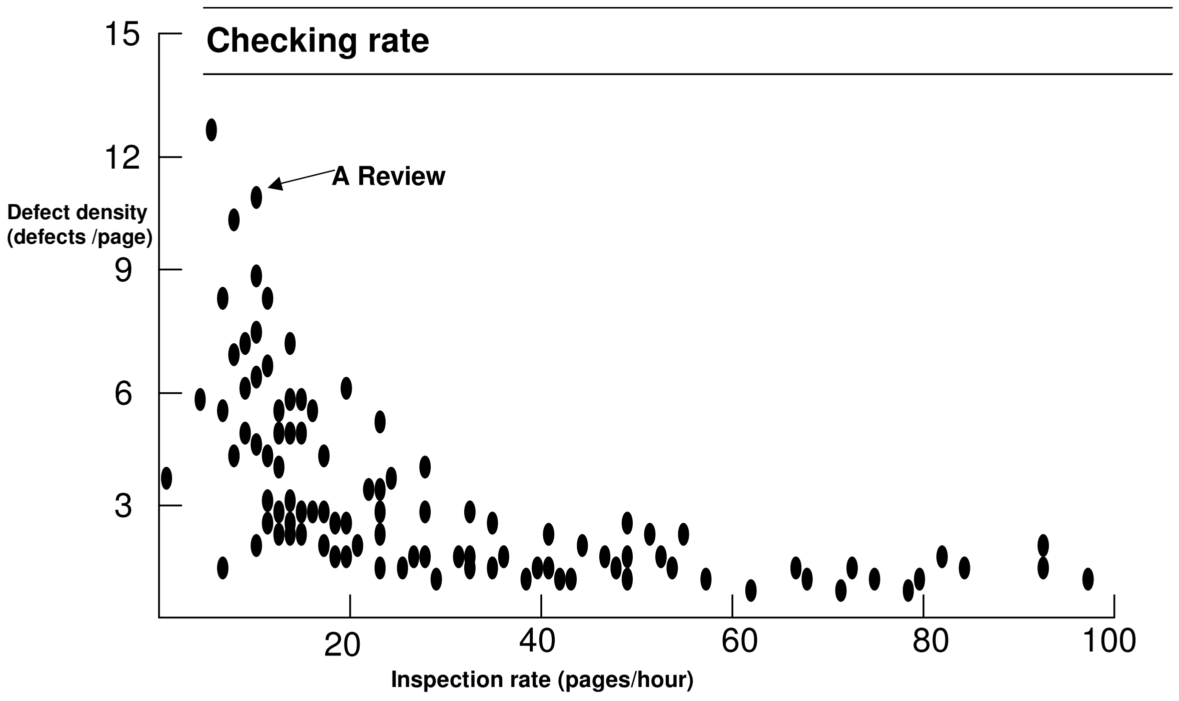


Figure 5 Checking rate optimum

The optimum lies somewhere on the 5 to 10 pages / hour for this organization.

If metrics are collected, use them to evaluate and improve the process, otherwise don't collect them at all!

3. TESTING

3.1 The V-model

The traditional waterfall model has a disadvantage: testing is put at the end of development. That's why the V-model was invented.

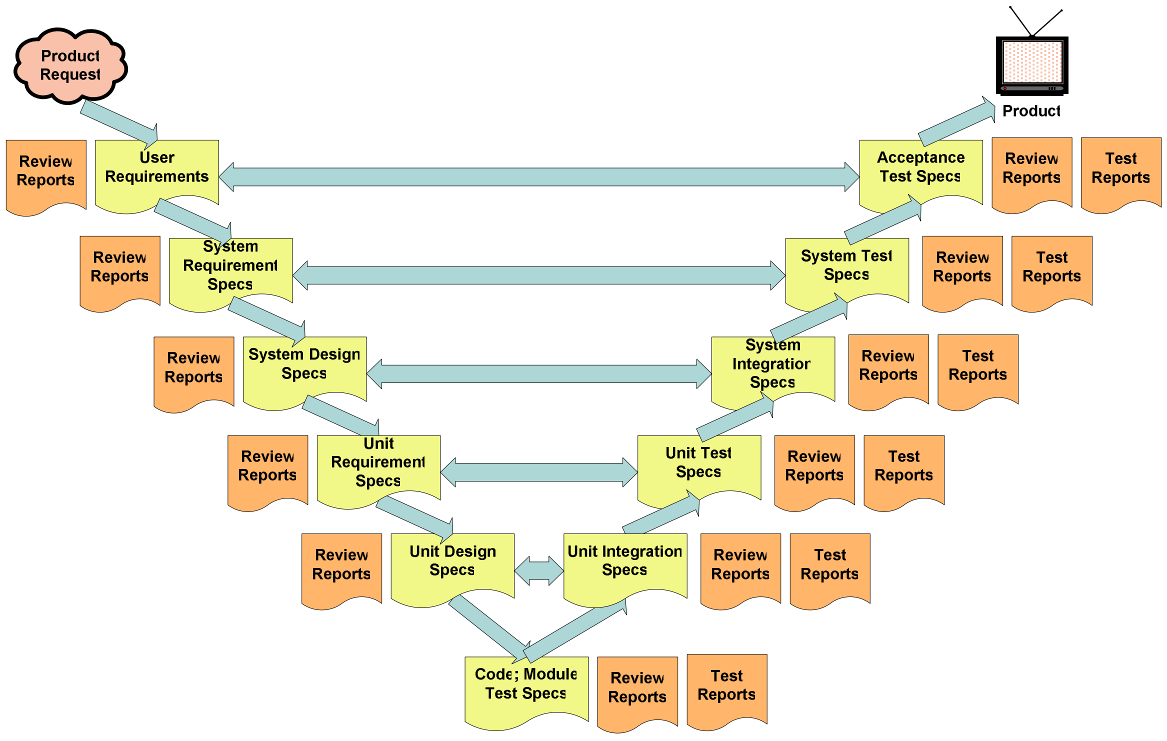


Figure 6 the V-model with its review and test reports

The system above is split up in several units. Every unit is split up in several modules. Dependent on the architecture, more levels can be added or removed. A simple system may contain 1 or 2 levels. A multi-disciplinary system has in parallel to the software part, also a hardware and mechanical part.

After the user requirements are mature enough, one can start writing the acceptance test specifications and the system requirements in parallel. The acceptance test cases prove the testability of the user requirements. This mechanism of writing in parallel the development part en the test part can then be executed for every level of the V-model.

3.2 Test process

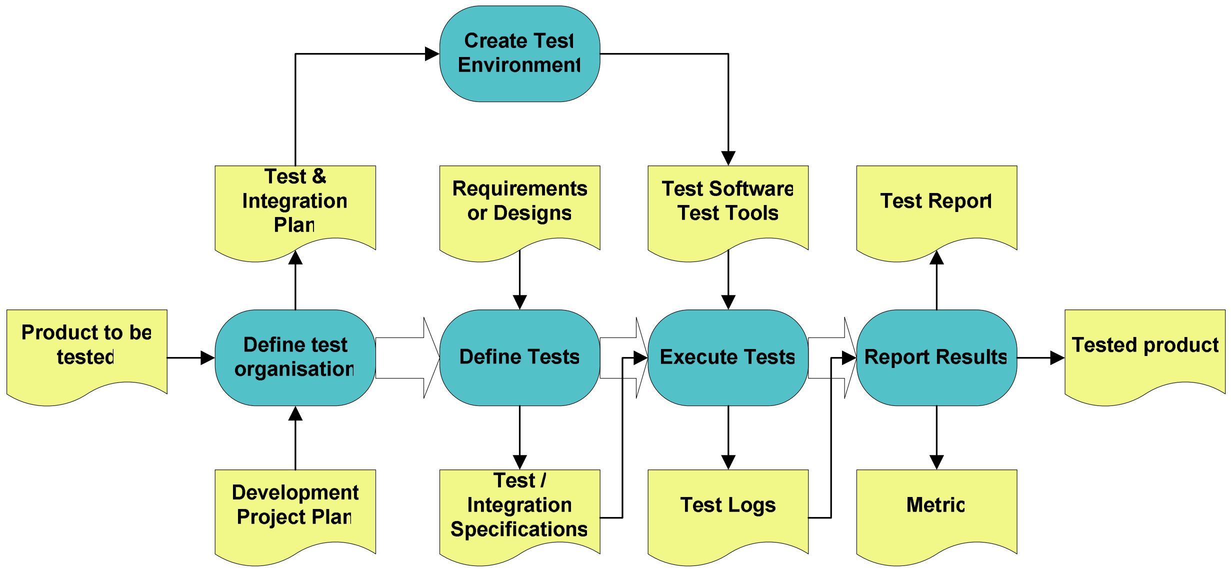


Figure 7 the test process

3.2.1 Define test organisation

As soon as the project has its product or user requirements and starts making its project plan, the Integration & Test plan can also be written. Combining the Development plan and the Integration & Test plan into one document is also possible.

Don't think that everything has to be defined in detail when starting to write the plan. Many things can be still unsure or open. State these open issues at the first page of the plan as a sort of action list. Agree with the customers that these open issues will be closed on committed dates.

3.2.1.1 Integration & Test plan

The output of the "Define test organisation"-step is the Integration & Test Plan. The content of such a plan should at least cover the following:

1. What is the Integration & Test scope? What part of the V-model is covered? E.g. system tests and unit integration only? Or all the way down the V–model (module testing)?
2. How is the test part organized? E.g. separate test teams? Or per development team one tester, or …. To who is reported?
3. What are the tasks and responsibilities of the tester(s)? E.g. also reviewing specifications, integration responsibilities
4. What will be delivered? Which Integration & Test specifications, which test reports.
5. Which resources/tools are needed? E.g. certain test tools but also which human resources. How many test systems are needed at what point in time and for how long?
6. What test strategy will be used? For more information, see 3.2.2.
7. What are the acceptance criteria (or entry criteria) if objects are taken over to be tested from development teams or from external parties? For instance deliverance of module test reports.
8. When is testing stopped (exit or pass criteria)? What is agreed upon with the customer?
9. How is the Integration & Test planning (work breakdown structure) in relation to the projects planning? Activities, milestones and resources.
10. What are the Integration & Test risks and what are the preventive actions? E.g. target system late available or untrained testers
11. How is regression testing handled?
12. How are problem reports handled?
13. Which test metric is assembled, why are they gathered and how are they used. For examples, see 3.2.5.2

Be sure to make the Integration & Test plan together with the testers. They are the ones that have to follow the plan. Use for instance several brainstorm sessions to get the above-mentioned topics sorted out. Don't fall into the trap of being the Integration & Test team leader that makes the plan all by himself behind the computer and assuming that the team will commit to it. Of course the plan must be in line with the development plan, so be sure to involve the project leader in the whole process.

3.2.2 Risk based testing

It has already has been said before: you cannot test everything in detail, so choices have to be made. A good technique for that is called ‘risk based testing. The stake-holders are asked what they see as risks for the system, and the depth of testing is adapted to those risks.

The technique contains the following steps:

1. Define for each quadrant of the risk matrix the testing methods (see section 3.5 for an explanation of these methods).
2. Discuss the technical and business risks per new functionality with the stakeholders and put them in a risk table
3. Give each risk a level (e.g. 1 = low risk, 3 = high risk) in the risk table
4. Calculate the total score per risk both for technical and business.
5. Draw the risk matrix from the total score (4 quadrants)

An example:

Step 1: Risk Matrix Quadrant definition

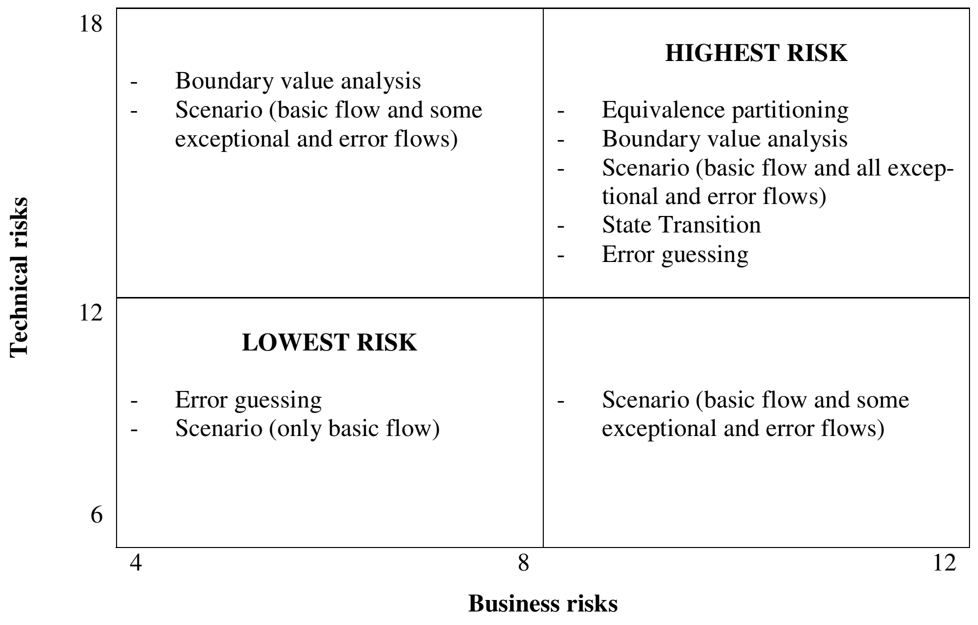

18
- Boundary value analysis
- Scenario (basic flow and some exceptional and error flows)
HIGHEST RISK
- Equivalence partitioning
- Boundary value analysis
- Scenario (basic flow and all exceptional and error flows)
- State Transition
- Error guessing
Technical risks
12
LOWEST RISK
- Error guessing
- Scenario (only basic flow)
- Scenario (basic flow and some exceptional and error flows)
6
4
8
12
Business risks

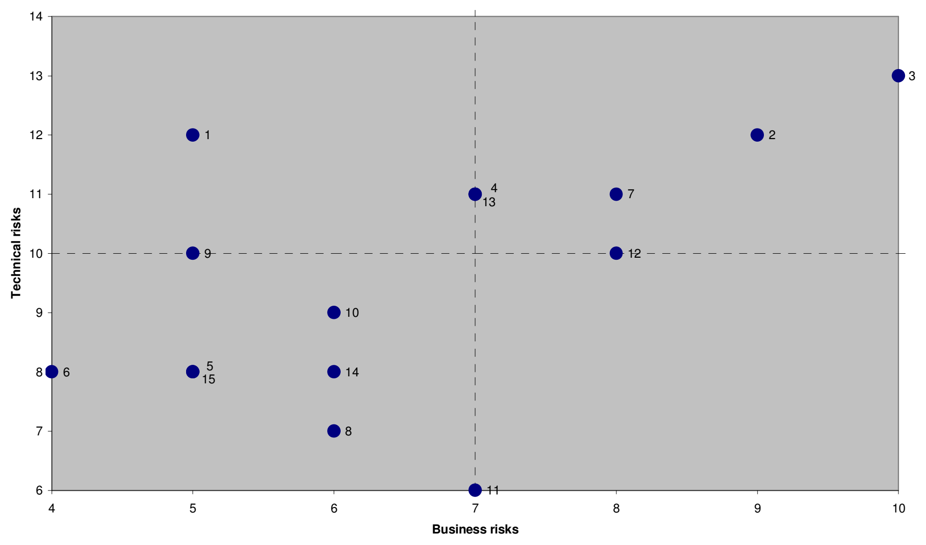
Figure 8: Test design techniques to be used in the different quadrants of the risk matrix

Step 2 – 4: Risk table

		Technological risk							Market/Business risk				
	Factor	Com-plexity	Size	Inaccuracy and inexperience development team	3rd party involve-ment	New develop-ment	Quality of specifica-tions	**Total score**	User impor-tance	Patient safety	Usage intensity	Liability (financial damage)	**Total score**
Id	**Item (new functions)**												
1	QVA	2	2	2	3	2	1	**12**	1	2	1	1	**5**
2	Full 3DRA support	2	2	2	2	3	1	**12**	3	2	3	1	**9**
3	2k2	3	2	1	2	3	2	**13**	3	2	3	2	**10**
4	UI enhancement	2	2	1	2	2	2	**11**	2	1	3	1	**7**
5	Auto config FSC ACP	1	1	1	2	2	1	**8**	1	1	1	2	**5**
6	QA performance	2	1	1	2	1	1	**8**	1	1	1	1	**4**
7	Rectangular Images	3	2	2	1	2	1	**11**	3	1	3	1	**8**
8	Portrait/Landscape export	1	1	1	1	2	1	**7**	1	1	3	1	**6**
9	MIP 3.1.net	2	2	1	2	2	1	**10**	1	1	2	1	**5**
10	Frame selection	2	1	1	2	2	1	**9**	2	1	2	1	**6**
11	File restructure	1	1	1	1	1	1	**6**	2	1	3	1	**7**
12	Redesign monitor LUT	2	1	1	2	2	2	**10**	3	1	3	1	**8**
13	Interoperability 4.1	2	2	1	3	2	1	**11**	3	1	2	1	**7**
14	Archiving annotations	2	1	1	1	2	1	**8**	2	1	2	1	**6**
15	Regression	2	2	1	1	1	1	**8**	1	1	2	1	**5**

Table 2: Risk table example

Step 5: Risk Matrix



3.2.3 Define Tests

Let’s first look at the input of the “Define Tests”-step: Requirements.

3.2.3.1 Requirements

Garbage in, garbage out

If requirements are bad, the test cases won’t be good either. How to write good requirements is outside the scope of this book so only a summary will be given here. Good requirements are:

• Concise	Keep it short
• Complete	Stand-alone readable and understandable
• Consistent	No contradiction with itself or other requirements
• Correct	Flawless description of the desired functionality
• Implementation free	Specify “what”, not “how”
• Unambiguous	Must have one interpretation only
• Verifiable	Fit to measure
• Achievable	Possible to implement
• Necessary	Must add value to the product
• Prioritised	For order in implementation and testing
• Unique labelled	Uniquely within the product to allow tracing

Requirement coverage

One of the main issues is requirement coverage. If all the test cases are executed, which part of the requirements is covered? A simple technique to make this coverage visible is a traceability matrix. An example is given below.

Req Id	Test case Id	Comment
UR01	TC01	
UR02	--	Not testable, see Problem Report 1253
UR03	TC02	
UR04	TC05, TC6	
UR05	To be defined	See Open Issue OI004

From the 5 requirements, 1 was found to be not testable, so a problem report was written for it. For one requirement (UR05) a test case is not yet written, so an open issue was raised for that.

The technique is simple, but when the number of requirements grows a maintainability problem emerges if these tables are not automatically generated. Nowadays there are several requirements management tools on the market available (see Appendix C) that will handle this problem.

3.2.3.2 Test/Integration specification

The output of the "Define Tests"-step are the Test / Integration Specifications.
The differences between these two types are:

- Test specifications: are the Requirements implemented correctly:
  - Check if the system does what it is required to do
  - Look at the outside of (part of) the system
- Integration specifications: are the Designs implemented correctly:
  - Check interfaces between parts of the system, do they communicate the way they were designed to communicate?
  - Look at the inside of (part) of the system

Content of a Test Specification

The content of a Test Specification should at least cover the following:

1. The traceability matrix (see previous section)
2. Test environment needed: hardware, test software, test tools. Be sure to mention versions of hard- and software if that's important. If the test plan covers this, refer to the plan.
3. A description on how to execute the tests: e.g. must it run on the target system, or can it run in some kind of simulation mode. And if it can be simulated, how is it done. Must it be done manually, or are tests automated. How long does the running of the tests last? Can each test be run separately or are there dependencies?
4. Which test method (section 3.5) is used for creation of the test cases? A combination of several test methods is possible.
5. The test cases
   a. Each test case must have a *unique identification* within the document.
   b. State the *purpose* of the test case. Give in a short sentence why this test case is needed.
   c. If a *setup* is needed for the test case (e.g. reset the system), then state that also.
   d. The central part of the test case consists of a number of actions or tests. The *input* that the tester has to give and the *expected output* of the system is described. Think about the way it is written down: it must be unambiguous and clearly stated. Tester John should perform the same actions as tester Mary if they execute the test case. This is called "reproducible execution".
   e. The *output* must always be measurable. E.g. output on a screen, test results in a log file, a visible movement of a motor or an update of a database.

Content of an Integration Specification

The content of an Integration specification is almost equal to that of a Test specification, the differences are:

- The test cases refer to parts of a Design instead of to a Requirement.
- Part of the specification is an Integration scheme showing, which parts of the system integrate at what moment. See also sections 3.3.3 and 3.3.5.

If a test specification is to be made from a large requirement document, be sure to make only 1 or 2 test cases. Let these test cases be checked by several people so that you know you're on the right track. Then continue with making all the other 100 test cases.

3.2.4 Execute Tests

The tester will execute the tests on a test system by feeding the system with the input from the test case. He compares the output with the expected output of the test case. If there is a difference this can be caused by:

- A bug in the test case.
- A bug in the product under test.

During the tests, the tester logs what he is doing. Especially on complex systems where several testers make use of the same test system, this is essential. Many costly testing hours are wasted because a tester had left the system in an unknown state behind.

3.2.4.1 Test log

Outputs of the "Execute tests"-step are test logs. The content should at least cover the following:

- Name of the tester
- Test date
- Target test machine
- Items tested with software version
- Activities/messages which are relevant for other testers to see, e.g. "system crashes when using the DICOM application, do NOT use at this moment".

3.2.5 Report Results

When all tests are executed, the tester must write a report to let the project leader know that the product has been tested.

3.2.5.1 Test report

The content of such a report must at least cover:

- Summary giving opinion of tester to accept this product or not.
- Name of the tester
- Testing dates
- Target test machine
- Items tested with software version
- Test environment used, if it deviates from the one in the test specifications
- List of test cases with the result: Ok or Not Ok
- List of problem reports written for the Not Ok's

3.2.5.2 Metric

Several metrics can be gathered on the test results. Some examples are given below.

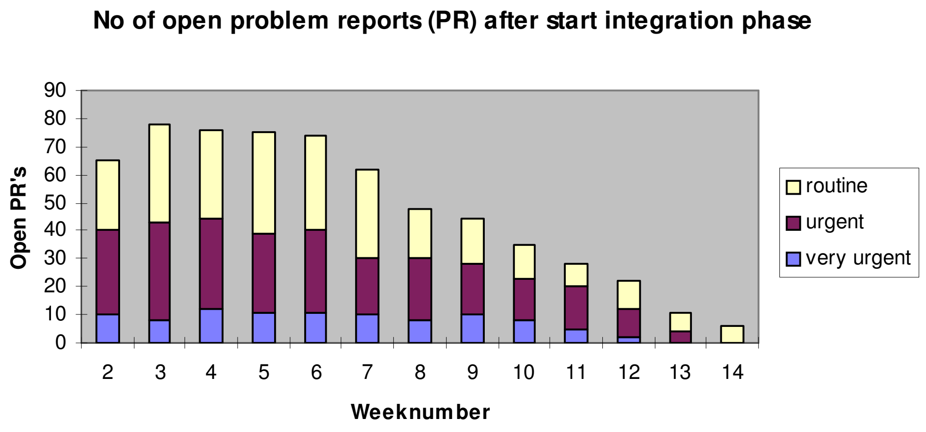


Figure 9 trend of open number of problem reports: purpose: progress on PR's during development of a product

In the above figure, the found problems after the start of the integration tests of the system are measured. The trend is looking good because the total number clearly diminishes and the very urgent and urgent PR's are down to 0. What are left are the routine problem reports.

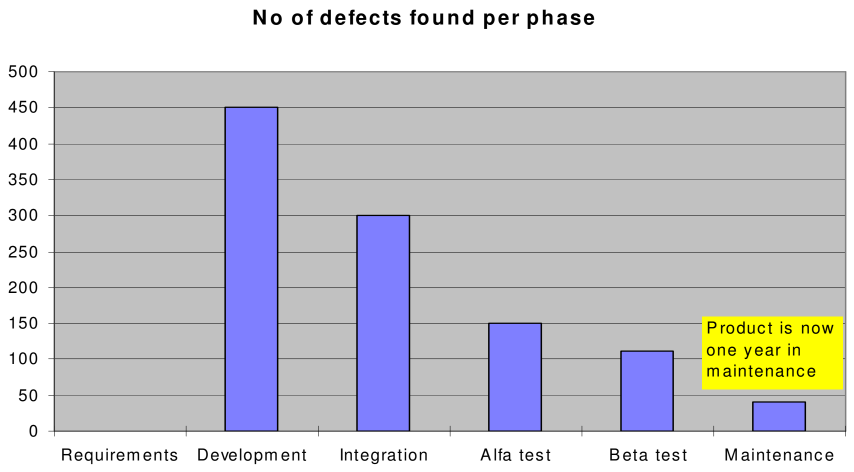


Figure 10 Number of problem reports per development phase; purpose: quality of the different phases of development

Above figure also shows a good trend. The number of found PR's is decreasing in time. But beware: the last phase (maintenance) is also the final stage of the product. If a product stays in the field for 10 years, the number of maintenance PR's will increase.

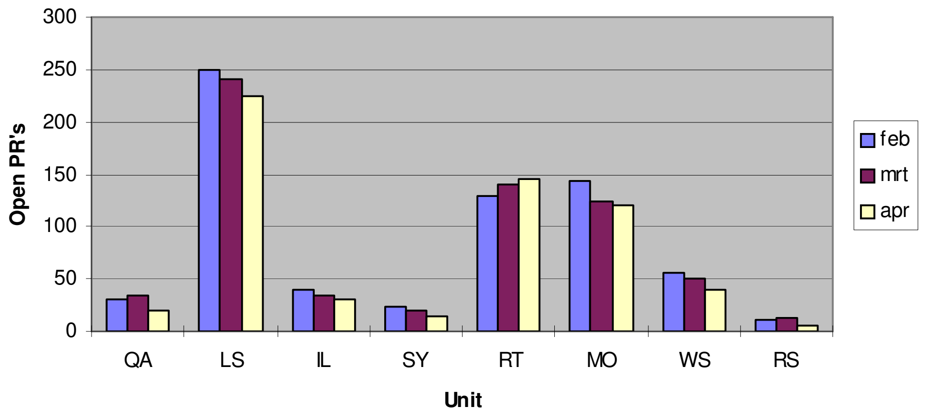


Figure 11 Number of open problem reports per unit per month: purpose: compare unit quality

In the above figure it looks as if the unit LS is of the lowest quality. What happened here, is that the responsible team of unit RS did not issue a PR for every problem they found. They just solved it directly in the code. The LS owners did a good job by consequently issuing PR's. Line management that saw these figures for the first time wanted to take measures on the LS unit instead of on the RS unit.

Metrics always have a history or reason behind the data. Find these reasons before jumping to conclusions.

3.3 Test phases

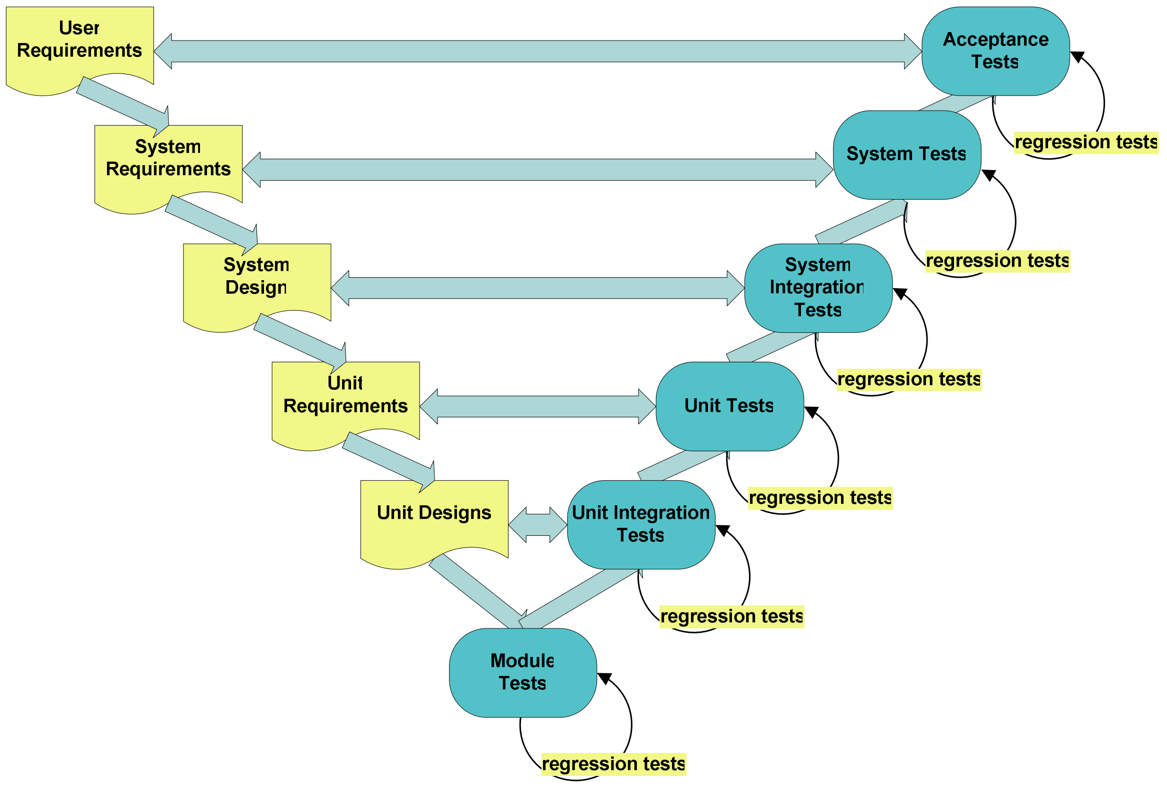


Figure 12 V-model and the several tests

3.3.1 Acceptance test

Why	Demonstrate to customer's satisfaction that system fulfills its original intended purpose. Defects may still be found, but the system should be robust at this point.

What
Formal acceptance of the system. The acceptance test specification is a contractual document and contains the validation of the user requirements (also a contractual document).
Typical questions to be addressed in acceptance tests:

- Can the new software be installed?
- Is the end-user able to understand how to use it effectively?
- Is it reliable, i.e., free of critical defects?
- Does it deliver acceptable performance?
- Will it work under "normal production" conditions in the end-customers environment?
- Can it be configured to meet the end-customers needs?

Evaluate completeness, consistency, and clarity of procedures and documentation from a user view:

- User-system interface, installation, training, and operations
- First meaningful test of user documentation with actual users

When
Planned during user requirement definition

Where	Execute in target environment
Who	Responsibility of customer
How	Only black box technique

3.3.2 System test

Why	Ensure all requirements (functional and non-functional) have been addressed satisfactorily.
What	Verify the (software) system requirements
When	Planned during software requirement definition
Where	If possible in target environment with little or no test programs, but always in a controlled environment.
Who	Responsibility of the project. Executed by an independent tester (test group), if possible, involve real end-users.
How	Only black box technique

3.3.3 System Integration test

Why	Check if unit/subsystem interfaces are implemented correctly.
What	Verify the (software) subsystem designs (architectural design). Not just check if system builds correctly. Check on inter process communication on unit level, e.g. • file/global data locking • deadlocks on semaphores • timely handling of critical interrupts
When	Planned during architectural (software subsystem) design phase. Let testers participate in project planning on unit development sequence
Where	If possible in target environment with little or no test programs, but always in a controlled environment.
Who	Responsibility of the project. Executed by designer or architect or integrator.
How	Do difficult/risk full parts first (e.g. integrate hardware early) Decide on integration strategy: • Big Bang • Incremental top down/bottom up • Back-bone: build up essential functions first: • I/O (network, display, keyboard) + Database Backbone can be used as platform for next subsystems to be integrated (incremental development)

3.3.4 Unit test

Why	Ensure all unit requirements have been addressed satisfactorily.
What	Verify the unit requirements
When	Planned during unit requirement definition
Where	Most of the time in a simulation environment. If possible, also use already other units (pre-integration).
Who	Responsibility of the project. Executed by the unit owner.
How	Only black box technique

3.3.5 Unit Integration test

Why	Check if module interfaces are implemented correctly.
What	Verify the unit design.
	Check on inter process communication on module level, e.g.
	• File/global data locking
	• Deadlocks on semaphores
	• Timely handling of critical interrupts
When	Planned during unit design phase.
Where	Most of the time in simulated environment with stubs and drivers.
Who	Responsibility of the project.
	Executed by module owner together with unit owner.
How	Only black box technique

3.3.6 Module test

Why	Check if module is implemented correctly.
What	Check for internal code errors and verify against unit design.
When	Planned during unit design phase
Where	Always in simulated environment (stubs, drivers and emulators)
Who	Executed by software engineer
How	Step 1: white box: reach code coverage target
	Step 2: black box: check functionality

3.3.7 Regression test

Why	Eliminate undesirable side effects due to software changes (change requests or problem reports)
	Ensure (part of) the system still functions during development: ensure a stable system.
What	Test changed part and all parts that interact with that changed part.
When	Done
	• On periodic basis during development
	• On interrupt basis during development and maintenance (after a software change)
Where	Performed on all work-products including the end-product
Who	Execution depends on work-product, e.g. regression test on system level done by test group
How	Must be reproducible at all times, input comes from other tests (acceptance, system, integration, unit and module). Candidate for automatic testing.

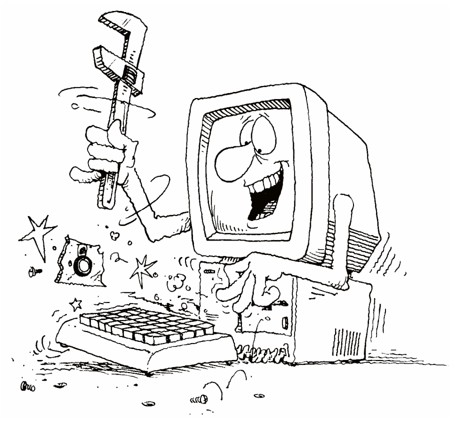
3.4 Automatic testing

Automating tests has some nice advantages:

- Possibility to shift (unattended) testing to 'non working' hours
- Testers are relieved from repetitive proceedings
- Test execution gets reproducible
- Enabling testers to concentrate on test results instead of execution of tests

However, it does not come for free. It takes time & money to make the automatic tool environment. Test code or test scripts must be written or an automatic test tool has to be customized to let it run on the application.

The automatic tool environment has to be tested also on bugs!



A fool with a tool is still a fool only making faster disasters

Where is the break-even point?

A rule of thumb says it pays back to automate if a test is expected to be executed 10 times or more during development and maintenance.

3.5 Test methods

3.5.1 Equivalence partitioning

Looking at all the possible input values, divide them into several classes. The assumption here is that every input value of a particular class acts in the same manner than any other input value from that particular class. So, in theory, it should be sufficient to take one value out of every class and write a test for it.

Lets look at an example of the testing of the logarithm function log(x).

Log (x)	**Output**	**Class Id**
-5	Invalid	1
0	Invalid	1
0.3	-0.522	2
0.5	-0,301	2
1	0	3
1.3	0.133	3
2	0.301	3
10	1	4
50.5	1.703	4
100	2	5
1000	3	6

Apparently, if this logarithm function must be tested, there is one class for the negative numbers, one class for the numbers between 0 and 1, and a class for every extra digit:

Input	**Expected output**	**Class Id**
Log (-5)	Invalid	1
Log (0.5)	-0,301	2
Log (1.3)	0.133	3
Log (50.5)	1.703	4
Log (300)	2.477	5
Log (5012)	3.700	6

The above tests are then sufficient for numbers up to 10.000.

Of course, the above example is very simplistic. In real life, a number of input values exist that must be combined to get one or more output values. And not every input value is so deterministic as the log(x) function. But there is your challenge!

3.5.2 Boundary value analysis

If all the classes are found, the next step is looking at the boundaries of these classes. The deeper thought behind that is, that many defects in programming occur at boundaries. Next the tests for the values are written:

- Just below the boundary
- Just above the boundary
- Exactly on the boundary

To get back to the previous log(x) example:

Input	**Boundary**
Log (-0.0001)	Just below
Log (0.000)	Exactly on
Log (0.00001)	Just above
Log (0.99999)	Just below
Log (1.0000)	Exactly on
Log (1.00001)	Just above
................etcetera	

3.5.3 Scenario

Developers test the products on a *functional orientation*: particular features are tested in isolation. E.g. in a word processor, all the options for printing would be applied, one after the other. Editing options would later get their own set of tests. This is a technical approach of the system.
End-customers however, use the product on a *task orientation*. For example, a very common task or scenario in a word processor is

- open a document
- edit the document
- print the whole document
- edit one page
- print that page

If only the strict requirements are followed on functional basis, the above task will not be tested. Therefore the requirements should contain scenarios.
In object orientation "use cases" (and instances of use cases called "scenarios") are very common. For each scenario a test case is written.

If there are no scenarios in the requirements available, talk to the end-customer or user of the application, together define the scenarios and let the end-customer write them down in the requirements.

3.5.4 State transition testing

If state transition diagrams are part of the requirements, the behavior of the components regarding these transitions can be tested. A simple example below gives the 2 test cases for testing the 2 states of a light bulb.

Test Case	**Input (event)**	**Initial State**	**Output (action)**	**Final State**
1	Push switch	Off	Light goes on	On
2	Push switch	On	Light goes off	Off

3.5.5 Error guessing

Error guessing is not a real technique or method but it can be very helpful.

Suppose there is a system of which is known that it has some weak spots in the communication between two tasks (due to the fact that this communication is badly designed: see section 1.3 “What is Design for testability” to really solve this problem). History shows that most problems always occur in this area. If all this is known, it is very wise to define a test that focus on this communication part.

A very familiar error guessing method is the monkey test. Sit behind the system and act like an ignorant user. Push all kind of buttons and keys and start all kind of applications. The target is to get the system crashed.

The monkey test has one pitfall: it is not very reproducible (or you must log exactly all actions performed).

3.5.6 Test attributes for non-functional requirements

Some Non-functional requirement can not be tested with the above methods. Here are some tips.

3.5.6.1 Capability

- Verify installation procedure
- Test minimum and maximum hardware configurations

3.5.6.2 Stability

- Test concurrent events
  - E.g. multiple users accessing one application at the same time)
- Test for memory leakage and other problems in long-running applications

3.5.6.3 Resistance to failure

- Recovery from power or hardware failure (warm restart)
- Security tests where applicable

3.5.6.4 Compatibility

- Interface with local and remote applications
- Interfaces with network services (e.g., transaction processor)

3.5.6.5 Throughput

Peak or average number of transactions (or other events) per unit time that the system can handle. Apply workload (usually with a simulator) to show that the system has the required throughput. Measure response time to ensure that response-time requirements are met as load increases

4. Appendix A: More historical software failures

4.1.1.1 AT&T long distance service fails: 1990

Switching errors in AT&T's call-handling computers caused the company's long-distance network to go down for nine hours, the worst of several telephone outages in the history of the system. The meltdown affected thousands of services and was eventually traced to a single faulty line of code.

4.1.1.2 Patriot missile misses: 1991

The U.S. Patriot missile's battery was designed to head off Iraqi Scuds during the Gulf War. But the system also failed to track several incoming Scud missiles, including one that killed 28 U.S. soldiers in a barracks in Dhahran, Saudi Arabia. The problem stemmed from a software error that put the tracking system off by 0.34 of a second. As Ivars Peterson states in Fatal Defect, the system was originally supposed to be operated for only 14 hours at a time. In the Dhahran attack, the missile battery had been on for 100 hours. This meant that the errors in the system's clock accumulated to the point that the tracking system no longer functioned. The military had in fact already found the problem but hadn't sent the fix in time to prevent the barracks explosion.

4.1.1.3 Pentium chip fails math test: 1994

The concept of bugs entered the mainstream when Professor Thomas Nicely at Lynchburg College in Virginia discovered that the Pentium chip gave incorrect answers to certain complex equations. In fact, the bug occurred rarely and affected only a tiny percentage of Intel's customers. The real problem was the nonchalant way Intel reacted. "Because we had been marketing the Pentium brand heavily, there was a bigger brand awareness," says Richard Dracott, Intel director of marketing. "We didn't realize how many people would know about it, and some people were outraged when we said it was no big deal." Intel eventually offered to replace the affected chips, which Dracott says cost the company $450 million. To prove that it had learned from its mistake, Intel then started publishing a list of known "errata," or bugs, for all of its chips.

4.1.1.4 New Denver airport misses its opening: 1995

The Denver International Airport was intended to be a state-of-the-art airport, with a complex, computerized baggage-handling system and 5,300 miles of fiber-optic cabling. Unfortunately, bugs in the baggage system caused suitcases to be chewed up and drove automated baggage carts into walls. The airport eventually opened 16 months late, $3.2 billion over budget, and with a mainly manual baggage system.

4.1.1.5 Deregulation of California utilities has to wait: 1998

Two new electrical power agencies charged with deregulating the California power industry have postponed their plans by at least three months. The delay will let them debug the software that runs the new power grid. Consumers and businesses were supposed to be able to choose from some 200 power suppliers as of January 1, 1998, but time ran out for properly testing the communications system that links the two new agencies with the power companies. The project was postponed after a seven-day simulation of the new system revealed serious problems. The delay may cost as much as $90 million—much of which may eventually be footed by ratepayers (belastingbetaler), and which may cause some of the new power suppliers to go into debt or out of business before they even start.

4.1.1.6 Ariane 5, June 1996

The maiden launch of the Ariane 5 rocket blew up 40 seconds from liftoff. The rocket and it's four satellites were uninsured and worth $500 million. The proximate cause of the crash was an overflow error due to an attempt to convert a 64 bit floating point value into a 16 bit integer. This error occurred in code that was non-functional after liftoff, when the error occurred. The Inquiry Board report provides details regarding the software failure and the design policies that lead to it. A note in the Risks Digest indicates that a complete system test would have found the problem but was vetoed for budgetary reasons.

In general terms, the Flight Control System of the Ariane 5 is of a standard design. The attitude of the launcher and its movements in space are measured by an Inertial Reference System (SRI). It has its own internal computer, in which angles and velocities are calculated on the basis of information from a "strap-down" inertial platform, with laser gyros and accelerometers. The data from the SRI are transmitted through the databus to the On-Board Computer (OBC), which executes the flight program and controls the nozzles of the solid boosters and the Vulcain cryogenic engine, via servovalves and hydraulic actuators.

In order to improve reliability there is considerable redundancy at equipment level. There are two SRIs operating in parallel, with identical hardware and software. One SRI is active and one is in “hot” stand-by, and if the OBC detects that the active SRI has failed it immediately switches to the other one, provided that this unit is functioning properly. Likewise there are two OBCs, and a number of other units in the Flight Control System are also duplicated.

The design of the Ariane 5 SRI is practically the same as that of an SRI which is presently used on Ariane 4, particularly as regards the software.

Based on the extensive documentation and data on the Ariane 501 failure made available to the Board, the following chain of events, their inter-relations and causes have been established, starting with the destruction of the launcher and tracing back in time towards the primary cause.

The launcher started to disintegrate at about H0 + 39 seconds because of high aerodynamic loads due to an angle of attack of more than 20 degrees that led to separation of the boosters from the main stage, in turn triggering the self-destruct system of the launcher.

This angle of attack was caused by full nozzle deflections of the solid boosters and the Vulcain main engine.

These nozzle deflections were commanded by the On-Board Computer (OBC) software on the basis of data transmitted by the active Inertial Reference System (SRI 2). Part of these data at that time did not contain proper flight data, but showed a diagnostic bit pattern of the computer of the SRI 2, which was interpreted as flight data.

The reason why the active SRI 2 did not send correct attitude data was that the unit had declared a failure due to a software exception.

The OBC could not switch to the back-up SRI 1 because that unit had already ceased to function during the previous data cycle (72 milliseconds period) for the same reason as SRI 2.

The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. The data conversion instructions (in Ada code) were not protected from causing an Operand Error, although other conversions of comparable variables in the same place in the code were protected.

The error occurred in a part of the software that only performs alignment of the strap-down inertial platform. This software module computes meaningful results only before lift-off. As soon as the launcher lifts off, this function serves no purpose.

The alignment function is operative for 50 seconds after starting of the Flight Mode of the SRIs which occurs at H0 – 3 seconds for Ariane 5. Consequently, when lift-off occurs, the function continues for approx. 40 seconds of flight. This time sequence is based on a requirement of Ariane 4 and is not required for Ariane 5.

The Operand Error occurred due to an unexpected high value of an internal alignment function result called BH, Horizontal Bias, related to the horizontal velocity sensed by the platform. This value is calculated as an indicator for alignment precision over time.

The value of BH was much higher than expected because the early part of the trajectory of Ariane 5 differs from that of Ariane 4 and results in considerably higher horizontal velocity values.

5. Appendix B: Code coverage tool examples

Below some commercial management tools are listed. This information is dated 17 July 2002.

TOOL: CTC++
Vendor: Testwell Oy
Category: Measurement
Description: CTC++ (Test Coverage Analyzer for C/C++) is an instrumentation-based tool for measuring test coverage and studying the dynamic behaviour of C and C++ programs. CTC++ is available in two packages: 1) as a host-based tool ("CTC++") and 2) as a tool, which facilitates coverage measuring in target platforms ("CTC++ Host-Target & Kernelcoverage"). GUI integration to Visual C++.
Date Posted: Jun 28, 2001

TOOL: C-Cover
Vendor: Bullseye Testing Technology
Category: Measurement / Test Execution
Description: Quickly find untested C/C++ code and measure testing completeness. C-Cover increases your testing productivity by showing you the regions of your source code that are not adequately tested. C-Cover measures condition/decision coverage to help you determine test cases that exercise the decision-making functionality as well as the computational functionality of your application.
Date Posted: Jun 11, 2001

TOOL: Rational PureCoverage
Vendor: Rational Software
Category: Measurement
Description: Rational PureCoverage is a powerful code coverage analysis tool designed for use by developers and testers during daily unit tests to increase software quality by preventing untested code from reaching end users. It is unsurpassed for ease of use and flexibility. With a single click, you can take advantage of an annotated source view that provides line-by-line analysis of either tested or untested code.
Date Posted: Jun 28, 2001

TOOL: TCAT for Java
Vendor: Software Research, Inc.
Category: Web Testing
Description: TCAT for Java is a test coverage analysis tool configured specially for Java applets and for use on Java-enabled browsers. Developers of animated Web sites can use TCAT for Java to determine that their Java applets are fully exercised by their test suites – a critical quality verification when Java applets support financial transactions on the web.
Date Posted: Jul 01, 2000

TOOL: ATTOL Coverage
Vendor: ATTOL Testware SA
Category: Measurement
Description: ATTOL Coverage assesses test efficiency by analyzing C, C++, Ada 83 or 95 code coverage information. Designed for both native and target platforms, ATTOL Coverage also supports compliance with the avionics' DO-178B standard. Source code that has not been covered (including dead code), which test covers which part of the code, and achieved code coverage information can quickly be visualized. When required for MC/DC testing, ATTOL Coverage displays information that help optimizing test case desig.
Date Posted: Oct 18, 2000

TOOL: TestWorks/Coverage
Vendor: Software Research, Inc.
Category: Measurement
Description: TestWorks/Coverage measures how much of your code has been tested. This powerful coverage analyzer does branch and call-pair coverage in a single test run and provides full support for all standard constructs and dialects of C and C++, using logical branch (C1), function call (S1) and path class (Ct) coverage.

With the new recursive descent compiler technology, it's easy to integrate coverage analysis into your standard "built-test-edit" process with a simple one-line change.
Date Posted: Jul 01, 2000

TOOL: LDRA Testbed
Vendor: LDRA
Category: Measurement / QA/Quality Mgmt / Reviews & Inspections / Test Development / Test Execution / Test Management
Description: The two main testing domains of LDRA Testbed are Static and Dynamic Analysis. LDRA Testbed's static analysis provides programming standards enforcement, complexity analysis and data flow analysis. Data Flow Analysis has been proven to be one of the most cost-effective methods of removing bugs from software. Dynamic Analysis involves execution with test data, through an instrumented version of the source code, to detect defects at run time and provide code coverage measurement. LDRA Testbed rep.
Date Posted: Nov 14, 2000

6. Appendix C: Requirements management tools

Below some widely known commercial management tools are listed.

- Doors (QSS): complex systems
- Requisite Pro (Rational)
- CaliberRm (Starbase): small systems

The link http://www.volere.co.uk/tools.htm contains an updated list of many more tools.

About the author

Willem received his bachelor's degree in Electronics in 1985 and started working as a software developer. During the first 9 years of his career, he has worked in projects for Océ vd Grinten, Organon Technica, Philips Medical Systems and Draeger and ended up as project / team leader.
In 1997 he switched towards the quality assurance and process improvement role in multi-disciplinary projects for companies such as ASML, Philips Consumer Electronics, Philips Medical Systems and Centric TSolve.
From 2001 onwards he started his own company Process Vision and continued to work as quality assurance / process improvement officer.
Next to that he conducts (from 1998 up to now) the software quality control course at the Technical University of Eindhoven as part of the OOTI program.

Revision History

Date	Revision	Comment
18Jan 2002	01	Initial version
29 June 2002	02	Added test methods and test phases.
3 July 2002	03	Added regression tests and automated tests
7 July 2002	04	Added comments from OOTI students 2002
15 May 2003	05	Added design for testability and review product roles Changed test plan into Integration & Test plan
1 July 2003	06	Added comments and explained questions from OOTI students 2003
16 April 2004	07	Prepared for OOTI course 2004; Added risk based testing; state transition testing and six-sigma results.
1 July 1 2005	08	Prepared for OOTI course 2005
7 August 2006	09	Prepared for OOTI course 2006
06 September 2006	10	Prepared for OOTI course 2007
23 July 2007	11	Small updates for OOTI course 2007