How to Save on Quality Assurance – Challenges in Software Testing

Sigrid Eldh[1]

[1] Ericsson AB, Box 1505, 125 25 Älvsjö, Sweden
Sigrid.Eldh@ericsson.com

Abstract. Quality assurance, and in particular software testing and verification, are areas that yet have much to offer to the industry. Companies that develop software need to improve their skills in this area to get the best return on investments. Important future strategies for survival are to collaborate with academia to find solutions to several difficult problems within software testing. Some of the areas and experiences in software testing that needs to be improved from an industry perspective is discussed, like test automation and component test. We have created a way to improve designers testing, which we call software quality rank. This ranking system takes known research results, including knowledge and tools from static- and run-time analysis and making them work in industry. The software quality rank aims to improve testing on the component level. We have saved a lot of money by improving the test area, and we share some of the lessons learned to aid other businesses with the same endeavor.

Keywords: Software Testing, Verification, Quality, Improvements, Test Automation, Test Process, Measurements. Software Quality Ranking

1 Introduction

There is a need for effective development in industry, since we are always in a competition with other companies for the business. Even if we are developing software for our own use, or regardless of what business we are in, we are always under cost and time constraints. We want the best quality in our software to the lowest price. Some organizations face legal actions; some lose customers – and the worst case is that you would go out of business. For most businesses, it is a constant focus on economy – to bring profit to the company. Nowadays, software is everywhere, embedded, and supporting many important functions in society.

The software industry faces many challenges: Software has to function, and be fast to the market, and deliver value in a competitive market. This has to be done with fewer resources. Since outsourcing is becoming more popular, both geographical and cultural diversity must be managed. Most software industries have problems with achieving good quality, reasons are: too few people or not motivated personnel, not enough skills, insufficient tools, and not the right internal and external support. Now

it is no longer economical to develop all software yourself, you buy commodities like operating system, database systems and administrative systems from a variety of third party vendors. These different systems do not necessarily work together and follow exactly your company's procedures and processes, and they often need tailoring to be fully utilizable. Most software producing and consuming companies have their own IT department, and their own tailoring of the systems they use. Whatever software you own, have customized, developed or used, you still need to save on maintenance.

Bad quality can have a lot of consequences. Can we trust the results; can we trust the software being reliable? What problems is a consequence of this software? Some consequences are even life-threatening, qualifying as safety-critical software. Developing bad quality software means an expensive maintenance, and extra support and administration (to compensate for bad software) which costs money to set up. Yet, many have turned the support into a money-making affair, a deal only possible where there is poor competition.

To understand what quality you need to have in your product, you must establish what a reasonable quality level is for you. One important question is: How long does your software product exist on the market? This will determine the level of investment in software testing. Experience show, that software has longevity much longer than originally designed for and expected. Often in the 10-20 years bracket, where people expect it to be rather in the 5-8 years. The investment goes far beyond the actual cost of the software creation. It focuses on learning, knowledge, invested and dependent data (used in the system), cost of transfer etc. Usually, there must be very superior software with a reasonable price to make persons change from one to the other. Instead, people tend to improve what they have, or find suitable work-arounds.

Another quality question is on the understanding of *what makes the quality in software*. There are many contributing factors, good requirements, well defined business, good business case, a strong architecture, having buy-in from all stakeholders, having a well defined, documented and reviewed code, using a good process with milestones, having and using supporting processes and tools, such as configuration management etc. In this paper, we will focus solely on one part of the research and development area – testing and verification. The fundamental question is: How do you test and make sure that the time you use in development and testing is effective and efficient? There is no easy answer.

2 Software testing is an expanding area

Software testing and quality is one area in the software development phase that still is not completely explored and understood. To understand, we need to cooperate with researchers in the area, but also cooperate with other companies facing similar problems. The competence, the know-how, of how to develop quality software, does not come easy. This competence will be a differentiating factor in the future. By understanding development and test, building quality software, and having proper and

effective testing practices, we will better understand how to avoid expensive maintenance. The problem is to find the right balance – how much risk can we handle, how much faults do our customers tolerate, and what expectances do we have on software? The main goal if we are producing, procuring and using software is that we would like to find the majority of the problems and failures before we put in into use, or, for that matter, deliver it to a customer.

We must understand where we should spend our effort and what the right works to do are. Is it worth developing software, if we do not know if it works? There is only one solution; we must test it to demonstrate it works sufficiently. What is sufficient testing? This is a basic and yet difficult question where numerous books [1, 2, 3, 4, 5, 6] try to aid in answering this questions. To save time one can summarize it to - *risk assessments*. What is important and must work is also a priority to test. We must also measure costs. Is it cheaper to automate our regression suite, than do it manually over and over? Is it cheaper if the designers spent more time in testing before delivering to an independent test unit? Who will find most faults and failures that matter for the usage of the product, the independent tester or the individual designer? Do they both find important faults and failures?

3 How does quality become a part of the software?

Understanding how quality comes into the software is an important to take leadership in your business. The first step of improvement is to teach test and testing not only to testers (who surely could improve their skills, but probably know more than the rest of the company). It is important that designers know how to test, so they could efficiently test that their own code and design the code, so it is easier to test. It is important that project leaders know test, so they can understand the process, the signals and the feedback during the development and testing. It is important that managers, and product managers understand testing, so they can determine the cost benefit of investing in a tool for automation, and the advantage of having a working product to show the customers.

To be able to master the important aspect of having quality in the product, means that a company must constantly develop and learn new skills. Improving the test team means, they can be more efficient in utilizing tools and designing test cases that finds problems. Working in an environment that tries to become more efficient and effective and constantly improve, is motivating on a personal level. Yet, many companies may need to be motivated themselves to understand the need of quality in the product, which has to do with the general maturity and quality attitude of the company. So, how do you handle and respect the testers and the quality work in your company and organization? Do managers really listen to the quality feedback; do you listen to your testers? Usually, testers will give you “the other side”, the problems, the troubles, the not so good and not so effective parts of the software. This is very important information, since it gives you a chance to act upon it.

By teaching about testability to system architects and system designers who are early in the development lifecycle, the system are designed a bit easier and faster, which means cheaper. Since we need to trace and debug all our software, we build that in from the beginning, as well as test frameworks. This will save us time in the maintenance phases, and when we re-use system parts, and update them. We invest in improve the testing skills for the testers, which will then find failures and problems more effectively. Ericsson has already reached CMM level 3 and 4, and does several CMM5 practices. CMM from Humphrey and SEI [7] was big (new) in the 90s, where also Ericsson had a large program, but we have now moved on to a more integrated view, where we recognize the benefits of CMMI, but it should not be external processes that drives Ericsson, but Ericsson driving the processes. Processes are not the major hurdle as of today. Today the focus in understanding steering and measurement principles, and what differs our business from others. Processes are a necessity when there is a big turnaround of people. Currently the retention is less than 1%. We are more interested in R&D of being more effective. Ericsson is a mature company, following all major quality certifications. In any large company, there are occasionally "bad" projects, but, there are very rare, when we are becoming more skilled in what is needed to develop good quality software. It is easier to select wrong business choices, than that the software does not fulfill quality aims. To be successful and efficient, one must have the core of the company working, the skeletons, like administration, configuration management, information management, computers, programs, tool and support alongside skilled and motivated people. The list is long. By having balance, order, clarity, and constantly measure, assess and evaluate the software, we are constantly improving.

4 Testing within Ericsson

Ericsson has a long experience with thorough testing, and test has status and is respected. Test is a complement to the Quality Assurance, but is also a major driver in the projects. We can have 10-20 levels of testing before reaching the final telecom operators, but occasionally it is less, depending on if you are in an application or in a platform. On the improvement side, which often happens to test intense organization is that we have a tendency to test quality into the product. We are now (with the help of SQR) slowly moving to building the quality in at an earlier phase. This saves us in time, but also increases the quality.

In Ericsson, there is invisible line - testers' finds trouble reports and the designers fix them. This means that occasionally it can become a gap in knowledge (trouble shooting skills) with the testers, and the testing view for the designers. Lately our new projects have solved this by mixing developers and testers together. This is a pendulum that swings, because if testers sit too long with the designers, they become "home blind" and will not objectively test the same way. So, in a few years, the separation will happen again. It is important that there always is one testing level and team that is independent in the testing (as independent as possible) to assure a good quality product.

Testers regularly use test tools and automates test cases. Testing is an area that has resources, and all see the necessity of doing so. If the code delivered from design has bad quality, the testing will take too long time and be too costly. To avoid this we use many mechanisms. Since we have large complex system, the sheer amount of software code and functionality is large. Therefore, it is important to divide the system into smaller and graspable parts, which also speeds up the troubleshooting time. The software is organized in levels, which are the first the *component test* (designers/developers own test) and low level integration. Then is a series of *independent test*: Integration Test, Functional Test , Sub-system Test, System Test, Node test, Application test etc, which are all separated in teams, goals, contexts, and test cases, but is often coordinated in larger projects by test coordinator. We are using a Test Leader for every level or team. We have often specially designated and skilled people for configuration management (also test configuration management), and other special areas (performance, upgrades/installations). All test leaders and test managers have a strong cooperation with project leaders, system architects and designers. Most independent test team uses a test process, and most templates are based on IEEE Standard 829 for Test documentation. These templates are adapted and specialized for our needs (adding measurements etc). As all aspects in testing, the test process is undergoing constant improvement. Almost every test team is using good test tools, either developed in house to suit the needs and fit the often proprietary environment. Test automation is a constant focus and a necessity. For every change and correction in the systems, we use regression testing as a standard practice. Lately the Software Quality Rank (SQR) has become a focus and target, which aims to improve the testing at the developers own area. SQR has become a de facto best practice within Ericsson, and saves a lot of time and effort in our systems.

5 Software Quality Rank (SQR)

The main reason Software Quality Rank [8] was created, came from the test managers needs. Test are often separated in the organization from the design, and have not enough impact on the quality that is delivered from design and development. The test depends on the quality that is delivered from design, especially if there are quality criteria that need to be fulfilled to be able to release the product. The main reason is that designers are measured and planned based on the functionality and the time available in the projects. If quality is to be a part of the product you can test it in, but that is much more expensive than to try and build it in earlier. So, the focus from the test organization must be to have a total life-cycle view on testing, i.e. the entire development phase. The need to improve quality must start early, and the main focus of the software quality rank was to focus on the persons that manifested the faults, the designers. In Ericsson, designers do have a large impact on the design and implementation of the actual software product, and by this, also have a large impact on the quality delivered. By targeting the designers own testing, their low-level component test (or unit test), this would not only teach good testing practices, but also make the designers aware of the quality aspects. Also, the current universities are not

teaching fundamental testing practices in conjunction with programming, thus, there is a great need in teaching developers how to test their own code. Another reason for improving was that too many failures were found too late in the software testing phases or after release. Another reason was that the failures were found by many people at the same time, which indicates that the cost of analyzing and administrating faults had a large overhead, but did not necessarily result in many of them being fixed. We also did a root cause analysis, and concluded that many of these faults could have been found in component tests.

We can simply look at Barry Boehm[9], who have done a large investigation on corrections costs, and seen that the later a fault is found, the more costly (almost double for every phase) it is. It is of course cheapest to find faults on the requirement phase, which also indicates that and good reviews and a controlled handling of requirements would lead to better quality. But, it is astonishing to notice that when the code "leaves the developers desk" to just be integrated with other software, the code almost doubles, and the cost when testing starts is also much higher. This points to that whatever effort the designer puts in extra, (a week extra test on designer level) could save as much as 2-3 weeks in integration and system testing.

5.1 Why should we test?

The true reason for testing comes from experience data and facts. Finding the faults that leads to failure saves money. It is important to know how much a true customer fault costs the organization to fix, to be able to count savings in investments in the development phase. There is estimations on the average cost of a customer failure to be between 1000 – 50 000 Euros, where average is around 10 000 Euro. Even in a best case, a fault would costs 100-500 Euros, just in time and the administration. However, compare these figures with the cost of trouble shooting, which is what most non-testing organizations do. Is troubleshooting better than testing? No, it takes 4-20 times longer to actually find the reason for the problem compared to if you actually executed a specific test case that failed. These figures are not general, and are depending on complexity of system and the systems debugging and trace facilities. We tend to forget that the overhead costs of handling, tracking, prioritizing, re-planning, re-testing is very costly, but necessary for any serious software vendor. In conjunction with this, you have the knowledge and skill question. If you are ever to become better, you need to learn from each failure not to repeat them again. This could be a costly matter too.

5.2 What are the reasons for bad quality?

The main problem goes beyond the obvious fault. We need to understand why it is "bad" quality in our software. Therefore, the practice of root cause analysis should be a part of your companies' improvements. You need to find the real reason for why you do certain types of faults. Often knowledge and complexity is a factor, you have unclear knowledge of how others use and utilize the software, which means you do

the wrong assumptions on how things work, how other parts of the software works and how users will actually use your system. A second problem for developers is that they are not totally aware or in charge of how your software integrates or is built etc. It means that the actual software that executes looks very different from the code they wrote. This is especially the case when modeling, or generating of code from 4GL languages occur. We found to our surprise that in complex, and hardware close software development, many designers had not sufficient or inappropriate test environment to be able to test their produced code at all. Some used simulators as the only support, which could be cost efficient, but is much depending on the quality of the simulator.

Other root causes from a developers perspective is plainly bad requirements and design. The software can never be better then the context it works in, and good quality is often strongly connected to the architecture used. The most important reason, above all other is that most often software is not tested thorough enough. This means if you test shallow you might stumble across faults, and assume that the system is ok, when it really isn't. Releasing the software then could be a disaster. You must improve you testing, so it would be a good measure of the actual quality you have in you system. The best remedy for any bad software is to do a "Root Cause Analysis", and then create your own "checklist" of the improvements you need to take, and prioritize them. For some system that means that more than one phase in the software life-cycle should be improved, not just the component test. It is important to understand that Ericsson have a mature functional and system test in place, the component test was a weak ling for us. Now we have targeted it, and the results have been great. This is the reason behind focusing on improving component test.

5.3 The Software Quality Rank Context

The idea about the software quality ranking is that it is aimed to handle most software regardless of programming language and domain. However, we learned that there are parts of the ranking that does fit procedural languages better, e.g. we measure control structure.
The goal was that the ranking must be flexible, e.g. should be able to measure software for testing registers, code in Linux kernel mode, third party software (bought code) and outsourced code, alongside the normal software, databases, system source code. Also, the ranking must be comparable which meant both measurable and have comparative characteristics, but also some area of interpretation, in the sense that the designers always owned the prioritization of what is important, targeted and to what extent.

The result is that Software Quality Rank became a way to decide what quality we should aim at, not only did it function as a "checklist" to hand over to the test, to determine what has been done and what has not. The ranking system became even more interesting for new code because code in production we already know the "rank" which is the actual quality (based on number of failures and faults found). What surprised us was that SQR became "political" within the company. It is obvious

that sometimes designers just don't know how to test, they have never learned in school, and there have never been any specific requirements other than "develop code". There are of course many developers that just do not want to test, and become a developer as a result of that. My advise to them is to change out of software, because if you are just producing code, and have no idea if it works or not, (hence, do not understand how to test it), you should probably be doing something else. The most common reason is that developer would most often love to spend time testing their code, but they have not enough time to do a proper job. This latter part is what I think we mostly solved with this system. As long as it does not become a goal in itself to have full scores on the SQR assessments, but instead use it as a communication tool to explain what has been tested, how thorough and what has not been tested enough. SQR gives a tool to talk about how much time is enough and what quality you would receive with what investment. We have also shown that SQR is economical: With small investments, you get significant improvements, in a phase and at a place where quality matters and that is in the actual product.

5.4 Understanding the software ranking levels

The software quality ranking is a term well used within Ericsson, where the goal is to identify steps to attempt improvement ladders similar to the CMMs five levels. The content with the component test improvement was that the Software Quality Rank was defined to focus the efforts on component test only. It that respect, it was totally different, since it was more focused on product quality rather than the process. We used the model with the following approach.

1. Rank 1 is baseline. All legacy code should be categorized and the list of the "good enough software" could be ignored, and all code with TRs should be selected and ordered based on how important they are, if they were central in the system and had faults, or if it was particularly fault prone, a component we often name as a "stinker" component. The reasoning is that if the code has worked, and has no faults, it is "good enough", and money should not be wasted to improve it. The major faults had already been removed earlier in testing. Thus, it should be important and be paid attention in maintenance if the code suddenly started to get TR's it might become a candidate for improvement. All NEW code should be improved, since here, no money has been wasted to test the quality into the code, so it was bound to be more fault prone than code that has already been tested.
2. Rank 2 is planning and selecting all new components, and a selection of modified components, and the most fault prone components – and put extra effort in testing, reviewing and measuring them, to hope that they are robust enough.
3. Rank 3 means that the component is analyzed, measured, reviewed, documented and tested enough, so that the designer *knows* and believes it is good enough, and there is proof to support that claim. At Rank 3 the component should be able to hand over to any other developer for maintenance.
4. Rank 4 moves from component to critical code sections, and the idea is that to rank 4 is only when you optimize, and require fault tolerance, you put extra effort in making sure these code sections within a component is reliable.

5. Rank 5 is for safety critical code, and could be compared to e.g. Department of Defense Standard 184b or other software code standards.

The goal should be that 95 % of the code in our products reaches rank 3, which is "good enough". Parts of the operating system etc., will probably have to be at a quality level 4. Telecom system does not have safety critical code, but companies who do would benefit from this sliding scale. Another purpose of having a scale, where most of our code never reaches more than half of the scale is purely psychological. From the saying, "If you aim for the stars you reach the tree-tops". Knowing that many do develop code to higher quality standards should be an inspiration. In the design, the ranking system was design general and for all software systems, which made the scale more reasonable.

5.5 The Areas of SQR

The focuses within the ranking system, and for each level of SQR, are on the following areas:

1. Reviews and inspections (they have long been proven to aid in improving quality)
2. Static Analysis, these tools are constantly improving, and can help qualify, measure and support developers to write better code and also look at their own code with different eyes.
3. Dynamic (run-time) analysis, which in this case refers to measuring coverage of the component tests, utilizing memory management tools, and supporting performance measurements. Dynamic analysis can also help in the message-passing, and stepping through executions, as well as measure other aspects of the code.
4. Dependencies and descriptions. This area generalizes many important factors that does support the quality implicitly. It defines that you need to understand how the context of the component is (its dependencies), it defines that you need to document (sufficiently) your component for others to have chance to aid you, but also to support your testing, and finally it talks about the need to have more than one person involved in the software.

The SQR scheme requires fundamental understanding of some concepts in quality practices. If they are not known, learn these principles about software testing from books, universities, conferences, courses or certification schemes, e.g. ISTQB, that also teaches you different test levels. For example, understand what a state-transition or state-chart is (also called finite state machines), and be able to model software (at some level), but not necessarily in detail. A state refers to a state in the model. State-transition lends them to create good test cases. Note that for each rank, all items on the earlier ranks should have been fulfilled to reach a rank.

5.6 Reviews in SQR

The first rank does not put any requirement on the reviews, since the idea is baseline. The minimum requirement is the overview review, where the components are qualified (based on their quality (if they are released) or new components, and then prioritized in a quality order. The component individual work for a designer is that if interfaces change, a peer should know new states' (assuming a state transition model) that affects others. In principle, no component should only have one designer assigned, but informal peer reviews (from another designer) are accepted. The second rank, the review targets are all interface parameters, where states should be documented (in a design specification) and the prioritized control flow known.

Rank three adds that the comments should be reviewed (qualitative, not quantitative) that they are sufficient, aid maintenance etc, design and test specifications (especially functional, data related) documented, state transition table or diagram shall be documented (design document level!). At rank four, the full state transition and demonstrated fault tolerance (no single point of failure) is documented (note, not for all code, but for the critical sections in the code that is involved). Attention is on critical components. Rank 5 requires a dept of documentation according to standard. I would suggest on the level of a course material (well known) how critical component works by designer and adjacent, Hazard analysis, and use some standard that fits the level of safety critical code.

That means that the reviews performed is on rank one a walkthrough, Rank 2 is the closest team, and then rank 4 it is important to bring in all stakeholders, such as testers to review the test specifications, architects to review the design documents, adjacent or dependent designers (maybe from other teams) to review the code and interface descriptions. Rank 4 and 5 are using Fagan Inspection method [10]. This implies that a review process must exist and be understood. Success in reviews are more common, if the reviewers have special focus, and comes from different roles, something that can be created with evolving checklists. The reviews are also to be measured (time spent, faults found, time prepared etc) from Rank 2, so fundamental knowledge of the review meeting have taken place can be checked.

5.7 Testing in SQR

Testing is the main focus in SQR. The first rank claim has two approaches. If you only create one test, that would be the "normal test cases" which would be what you would use as the regression tests. These tests are the functional test cases, traversing the component. At this initial baseline stage, there is no emphasis on testing fault handling, if that is not the component created. Rank two immediately place a strong focus on the testing. First you need to plan which of components you are going to put extra effort in that should be documented and followed up in a Component Test plan, or as an explicit statement in another design plan. Component names should be named and the rank they should aim at.

At rank two, testing fault handling is a must, as well as clearly define what are valid and non-valid inputs, and any delimiters that might exists, for example the maximal and minimal value for interface parameters. At this level, the testing starts with doing functional tests, writing test specifications for them, and then using the coverage tool support to complement the testing with structural tests (aiming to achieve coverage, and cover what you missed). The functional test cases mean on the component you perform black-box, data dependency test, and you start to automate for regression suites. You also make sure your component work in its context, by doing low-level integration tests, or moving software to a target hardware. Non-Functional test cases lie aiming at testing real-time or performance situation, are optional, and depending on possibility at this level. The structural test cases are automated when possible. Note that a structural test case creates the test case based on the structure.

Rank three aims to have at least 30% automated test cases of all test cases. This is actually a very low figure, and often hard to estimate. Rank three adds that the functional test cases – the regression suite – must be automated. You are also expanding the number of test cases, by adding data (not only equivalence class testing, but also boundary value testing. To achieve rank three easy, it should be possible to expand the input variables easy (no hard coded input to test cases) by having automated and improved dataset. The last testing is that more stress, duration, endurance tests should be performed, aiming to really work the component with both faulty and correct data. Underlying these ranks, is that you need to create a test tool harness environment, you need knowledge and know how of creating test cases, and also test automation, and you need a proper stubbing and support to be able to test your components in this manner.

5.8 Static analysis in SQR

The static analysis method is used for several purposes. Collecting static measurements, like lines of code (from rank 2) and complexity measurement is an aid for planning purposes (how long time things take), but does not give direct feedback on the quality. Especially complexity measurements are a truly debated subject, where the relation so far to complexity seems to be size (very small or very large). Good insight of these measurements can be found in Les Hutton's book "Safer C"[11] or Fenton, Pfleeger [12]. We have not experienced any benefit of McCabe's Cyclomatic complexity [13], but nestled calls (i.e. coupling) seems to have a connection with quality. We use – from rank one, the judgment made by the designer (rank 1-5 where 5 is highly complex) which seems to work for all our planning and targeted efforts. Static Analysis is often underestimated or overestimated. There are plenty of tools that most of them work as a second support of re-reviewing your own code. The problem is that they give to many "false positives" – claiming that it is a problem, when there really is not in a closer look e.g. QAC/QAC++, Jtest, Flexelint or Lint. Lately these tools have evolved. An example is C++ Test from Parasoft, that gives a test harness framework and also the possibility to create your own design rules and search for them. This is a powerful aid in a complex environment. Another tool,

Coverity, can find real dynamic problems by using information during compilation, where we in one product found 1200 faults, where 300 were real faults. Note that even if these are high numbers, the likelihood of these faults emerging to a failure in our fail-safe systems are very slim. Other measurements in this field is % comments (measured from rank 2), where no comments is not acceptable. This is a typical indirect measurement, since it is the quality and not the quantity of comments you are aiming at, and there exists not easy measurement method to say so, except human judgment. We are also measuring from rank 2, the number of test cases (in absolute numbers) to see the number grow, and also how many of the test cases and their variants that is automated.

5.9 Run-time analysis in SQR

The measurements in run-time analysis are strongly related to testing i.e. code coverage. At rank one we do not require other than a static analysis tool, so code coverage is estimated statement coverage. If a tool exists it should of course be measured. For rank 2 a tool measurement is a must, and the goal for statement coverage is to achieve 100% feasible code coverage. This means that all areas where it is possible to reach coverage you should aim for it. This might vary, and is not an average of many components, there might also be a need to prioritize within a component if it contains many files. Good background information here is that the normal cases in coverage usually reaches 50-70%, so a figure must definitely reach 85-95% to be considered tested. Rank three the aim is to achieve 80 % feasible branch coverage and at rank 4, 100% feasible branch coverage. At rank 5 you should reach 100% feasible MC/DC or Modified Condition Decision Coverage or the equivalent 100% feasible State Transition coverage. Other run-time analysis is profiling such as performance (where it is applicable) at this level. Other important parts is using tools for memory error detection & analysis e.g. Purify, TestRT, Bounce checker, TestExpert, which is a necessity in several programming languages.

5.10 Conclusions about SQR

The software Quality Ranking system have saved Ericsson months in system testing (90 % of system test was passed in a third of the time!). As a result the test cases were rewritten, and had to improve, otherwise they did not find any faults. Most of all we saved 67% of testing and administrative overhead, important time that now is used to improve the product. Not only Ericsson has had success with SQR, one safety critical system has used it to qualify the rest of their code, and a Swedish Bank have also have successes. What is important with this work is that it contradicts fact that low-level (component testing) is not beneficial to improve. It has had a major impact on the quality in our system.

6 Automation tools supporting test execution

As any large industry with complex software, and high requirements on robustness and reliability, many different test tools has been created, procured, utilized and discarded. Since Ericsson produces a lot of proprietary middleware and platforms, there exists a unique experience with developing own test tools that has grown out of a need to be effective and efficient, and to be able to regression test the software. Reuse strategies are common, with component thinking, and integrated system, which means the systems exists in multiple versions and in a large variety of applications, users and customizations. This puts high requirements on being able to verify the systems, and to have traceability to requirements. The test cases needs to be reused in different settings (they follow product) and the management systems are large and distributed (global) and are used to keep track of the different test case repositories. There are many parallel projects, and organization has large freedom to find and utilize tool environments to suits the context of the software. Most common schemes can be found, where Ericsson has their own development e.g. TTCN3, TCL, Perl. Because of proprietary operating systems and hardware, many tools are tailor-made to fit the environment. Most test tools are command and script driven, and very few capture-playback tools can be found. There are also a set of test harness tools at different levels of software. Using test tools are a standard practice at most levels of test.

6.1 The test automation challenge

Test automation is not an easy task. The test automation area needs to mature. Many companies fall into common traps like ¨Automating your manual test" or "We have 600 test cases, now we automate them", which are beginner mistakes. Some of the consequences are that the test suite will test the *same paths* over and over, which will have a consequence that only the part we know (i.e. is automated) have a probability to work. The coverage for the automation suite could be below 10%, and there is little evidence that the software is robust in those cases. Another consequence is that the programs are not utilized where it is good (and humans where they are good) to test, so the manual tests might be very difficult (time consuming) to automate (not feasible test cases). Another of the dangers with test automation is that it is easy to make the wrong assumptions, it changes our belief system. It the test automation did not find any fault, we assume the system is fault free. If we had run the same tests manually, we might have observed interesting (and potential malfunction) behaviors in other (dependent) parts of the system, that an automated tool could not capture. Many automate with the intention to save effort, but test automation could be very labor intensive if the system changes or is updated (where in some cases the automation cannot be reused and test case expansion is difficult). Often testers loose the overview of tests rather fast (management) and one might focus so much on the test execution, that other labor intensive stages of automation is forgotten, e.g. to automate input (test data) or output (logs).

6.2 Lessons learned – automated testing

Test automation can save a lot of money, and we have shown that even within the project time-frame we can save in development time [], where the break even usually is around the third regression. Contrary to general assumption, most test automation suites are *semi-automatically,* where only parts of the test case execution are automated. The degree of automation is probably dependent on the software created, where our systems are failsafe, hence system will recover even if a failure occurs and you need to observe it closely. Because we have complex software, most of actual preparations in test time are consumed by editing test suites. The problem is often not to automate the test case, but to determine if the test case was successful or not (did a problem or a side-effect happen? which indicates the *observability* of faults are low. This is of course not the case for all systems. Designing the test system is as important as designing the system, where attention should be made to test levels, libraries, test creation guidelines, and test case templates. Another experience is that seldom does one test tool solve all the testing problems. Some tools and test cases support one type of testing, another test tool supports another, and also different tools are used at different phases in the development cycle.

6.3 Designing test cases

One of the hardest practical problems is to create test cases that capture the product and not just focus on the requirements of the product. Requirements are always a small selective part of the product. Depending on the skills of managing defining, and classifying requirements the according verification can vary, if not design levels and component levels are captured in the testing. To understand and prioritize (risk assess) what is important is therefore a must for a selective and successful testing. Testing all paths is never feasible for applications that are more complex. The knowledge of testing and test techniques are often shallow, even for skilled testers. Most testers learn the system, how the system is used and how it works, and builds test cases based from those experiences. There are approximate 29 classes of test techniques and more than 70 different ways of testing, yet we have little guidelines for when a test technique is efficient, under what circumstances, and in what domains. This area still needs a lot of research and collaboration between industry and academia. There are much support (books, courses) to learn about the techniques, but not enough on practical application in particular types of systems. One difficult but fundamental principle is that test cases should be easy to expand (take any input data, preferable automated). An emerging research area is the understanding of the different Testing techniques to more efficient and effectively (and maybe also automated) being able to find faults faster. There is a need to understand what failures are important. We also need to understand better, what is structured and systematic testing. Ericsson works with universities to evaluate different test techniques and creating better guidelines [14]. This would aid to understand the different test design techniques and improve the test case selection. In other works, we need to improve and understand better how to select test cases and where and how to apply the technique to best benefit. We must also understand how one can automate the

technique or parts of the technique. Can one automate the test data as well? Other interesting areas for research and industry are: understanding the stopping criteria. When is testing enough? Testability in a system is also interesting, and questions like "What kind of coverage techniques do you have, and what do they find?" Another area is static analysis and also research intense is inspections and review. In the Software Engineering field, the possibilities are endless. Evaluating and clarifying problems must be a part of the common work in industries and research, and hopefully together.

7 How to save money by test

To understand how much money you can save by proper testing, you must first start to measure – so you can show your improvement. You need to start measure what costs and understand why different things cost what. A good example is to know the total cost of *one* fault that reaches the customer, including all circumstances, such as administration and labor in hours. What you do not know, make a qualified guess until you know better, and declare that guess. It is hard to put a price on lost reputation. To put a price on the test cost and benefit, one must need to understand what is important. Test does not only find problems, it also demonstrates that the system is working ok. Test teams bring valuable information, a measurement in itself, about the quality of the software. Companies need to see the benefit of independent testing of finding the faults yourself which moves the company into the need to prevent problems and be pro-active. Quality assurance is an area that has more to it than system qualities. It brings order in chaos; it sets the goal and the limits for investments. Much has to do with test maturity. One can easily see the improvements in a company, when the testers and test effort move to the earlier stages of development. Prevention is cost effective, and gives fewer faults, less maintenance as well as automating the test from the start. This is what internal efficiency is about. It saves money in maintenance and gives new customers an insurance of a quality level. The knowledge and know-how about testing is a company asset. You need early feedback on development, and testing gives you a change to act on problems. Improving is not easy, and you need to improve to be excellent, and do it with structure, people and motivation. To support your improvements you need a process to help you how you do things. Otherwise, you are facing the risk that the improvement and investment is random, and you cannot utilize the benefits. If you do not measure time, effort, success etc., you can never show improvement, and you can not really *know* the problems either. You need to know where you are and have a perspective. Here you can use experts from outside and assess where you are, with improvement models (for example TPI [15] or TMM [16]). If you do not know where you are going, you can take any path. To understand what you aim for you need to know where you are going and have a vision and goal for your efforts. For this, you need a test strategy that helps you for short term and long term improvements, but also for the entire scope of quality and testing.

8 Summary

The knowledge, skills and know-how is the key to get efficient and effective quality assurance and testing, and save money in development and maintenance. It is important to utilize the collaboration opportunities between Industry and Academia, through mutual exchange of problems, knowledge and solutions. Having a test focus in you development, having a test process, that you can constantly improve, understanding and measuring the different phases, in what order tasks are to be done, and provide templates for fulfilling them are some of the ways that is practical. Test automation is a very difficult area, but could save a lot of money if done correctly, so the automation suite is easy to maintain (which is often not the case). Another difficulty understands that not all tasks in testing are possible to automate, and automating test execution does not mean that the entire testing process is automated. It still needs skills and software competence. Whilst improving the test, do not forget or ignore the development and the designers testing. The Software Quality Ranking system can aid to put correct requirements, skills and effort into this phase. By constantly learning new test techniques, and expanding the ways of testing, much can be learned – and here is also a research are that still has much unsolved. Other areas for future research and better industry practices are the testability in the system. It means both how to build quality software, but also that testing starts early in the software life-cycle. Software Testing saves and creates money by contributing to the quality of the product. Software Testing is a Challenge, and will not be mature if industry and research take action in the field.

References

1. Kaner, C. Falk, J. Nguyen, H.Q.: Testing Computer Software, 2nd ed., VNR, International Thomson Computer Press, 2nd ed. Boston, 1993
2. Fewster, M. Graham, D.: Software Test Automation, ACM Press, Addison-Wesley
3. Beizer, B.: Software Testing Techniques, VNR, International Thomson Computer Press, 2nd ed. Boston, 1990
4. Myers, G. The Art of Software Testing, 2nd ed. John Wiley & Sons, 1979
5. Black, R. Critical Testing Process, Addison-Wesley, 2002
6. Whittaker, J. A.: How to Break Software: A Practical Guide to Testing, Addison-Wesley, 2003
7. Paulk, M et al. SEI - Software Engineering Institute: The Capability Maturity Model: Guidelines for improving the Software Process, Addison-Wesley, 1994
8. Eldh, S.: Software Quality Rank, ICSTEST, Dusseldorf 2004
9. Boehm, B. W.: Software Engineering Economics, Prentice Hall, 1981
10. Fagan, M. E.: Design and Code Inspection reduce errors in program development, IBM Journal No 3, 1976
11. Hatton, L.: Safer C: developing for High-Integrity and Safety-Critical Systems, McGraw-Hill, 1995
12. Fenton, N., Pfleeger, S.L.: Software Metrics a Rigours Approach, PWS Publishing, 1997
13. McCabe, T.J., Butler, C.W.: Design complexity measurements and testing, Communications of the ACM, 1989

14. Eldh, S., Hansson, H., Punnekat, S., Pettersson, A., Sundmark, D.,: A Framework for Comparing Efficiency, Effectiveness and Applicability of Software Testing Techniques, TAIC, IEEE 2006
15. Koomen, T.,Pol, M.,: Test process improvement: a practical set-by-step guide to structured testing.Addison-Wesley Longman, 1999
16. Burnstein, I. Suwannasart, T.,Carlson, C.R.: Developing a Testing Maturity Model: Part II, Proc. Crosstalk, The Journal of Defense Software Engineering, 1996