# Experiments with Component Tests to Improve Software Quality

[1,2]Sigrid Eldh, [2]Sasikumar Punnekkat, [2]Hans Hansson
[1]*Ericsson AB,* [2]*Mälardalens University*
Sigrid.Eldh@ericsson.com

## Abstract

*In commercial systems, time to market pressure often result in shortcuts in the design phase where component test is most vulnerable. It is hard to define how much testing is cost effective by the individual developers, and hard to judge when testing is enough. Verification activities constitute a major part of the product cost. Failures unearthed during later phases of product development escalate the cost substantially. To reduce cost in later stages of testing by reducing failures is important not only for Ericsson, but for any software producer. At Ericsson, we created a scheme, Software Quality Rank (SQR) as a means to improve quality of components. SQR consists of five steps, where the first is where the actual "ranking" of components takes place. Then a selection of components is targeted for improvement in multiple levels. Most components are targeted for rank 3, which is the cost-efficient quality level. Rank 4 is intended for code with optimizations whereas Rank 5 is the target for safety-critical code. The goal of SQR was to provide developers with a methodology that prioritizes what to do before delivery to next system test phase. SQR defines a stepwise plan, which describes how much and what to test on component level for each rank. It gives the process for how to prioritize components; re-introduces reviews; requires usage of static analysis tools and defines what coverage to be achieved. The scheme has been used with great success at different design organizations within and outside Ericsson and we believe it supports industry in defining what cost-efficient component test in a time-to market situation.*

## 1. Introduction

Telecommunication systems are large complex systems. They combine proprietary hardware, firmware and software, are often constructed with fault tolerant and fail-safe features, on distributed, self-organizing component based system often including several gigabytes of interacting software using multiple protocols. Testing these systems is a challenge, due to the low observability of failures, but also due to the sheer scale and complexity. The reliability of these systems is a great concern, since the customers demand high-quality systems. In our industry, reliability means achieving telecom grade quality, i.e. 99.999% uptime of the systems. Testing of these systems has historically involved multi-layered functional and non-functional testing, and we have felt that the focus on automated testing, regression suites and functional test approaches have in a pressured time to market left the designers own testing somewhat compromised. The time pressure in industry has changed the developers processes from careful implementation, desk-check, review and test in isolation to a coarser process, not explicitly defining what should be done other than "produce code". There are a lot of available approaches for component test involving both dynamic and static testing. At Ericsson, we defined a step-wise improvement model, that we call Software Quality Rank (SQR), to guide our design teams in component test improvements. We have piloted this model during an 18 month project, commencing 2003, where 9 design organizations with a total of 23 design teams from around the world participated. Now SQR is used both across Ericsson and at other companies and contributes to better quality. The experiences while applying these experiments and the result of the experiments, have given us insights in difficulties of deploying improvements.

The outline of this paper is as follows: section 2 describes what SQR is, the intent and the different steps in the SQR model. In section 3, we describe the experiments in the different development organizations. Subsequently, achievements and validation aspects is described followed by related work. We present our conclusions and directions for further work in last. We have written this paper in a fashion that will make it possible to use and utilize the concepts at other industrial settings.

## 2. Software Quality Rank

What is Software Quality Rank? It is an improvement program focused on the definition and improvement of component test for software developers. The basis of SQR is the idea that there exists a way to select what parts of the software that should be improved, and if a strategic choice of what components to improve is made, it will impact the overall system quality. SQR will take into account new code, legacy code and modified code.

### 2.1. Motivation

This improvement program is based on the assumption that some form of component testing is already done, since design organizations have delivered software that has been used commercially for years. The problem we are aiming to address is that too many failures are found too late in the different testing phases before delivery, which makes development costly. Testing quality into the system at the functional or system test level is possible, but the result is long delivery times since regression tests needs to be performed in addition to a lot of failure administration (e.g. analysis, debugging, correction, and fault prioritization and handling). The idea is to move some of the test effort from the test organization to the development and design organization. The extra time should then be spent on quality enhancement at component test, before handing over to the next test phases, such as functional and system test. We wanted the quality to be more predictable at an earlier stage, so that we could estimate better how much testing remained before release. We analyzed our failures and faults, and draw the conclusion that many of these faults could have been prevented at much earlier stages, when they could have been manifested. It seems logical that the lack of test education in the design organization has an impact on quality. Most universities lack education in software testing, except for brief introductions as part of software engineering courses. This is not sufficient knowledge required to perform solid testing. Many developers become developers because they do not like to test, and assume that that is what testers' job is. The most common cause to bad quality from developers is to our minds is that there is not enough time in the development phase set aside for quality improvements.

We believe that by introducing this improvement program we can support developers on what quality work they should focus on, [5], [6]. What is more important is that the selection of targeted components is done in a way developers have themselves chosen. This makes the developers committed, since have chosen based on what they believe is reasonable to perform within the time given. When developers have an efficient test environment, good tools at hand and the knowledge of how to produce quality software, they are more likely to do so. They become more "fault aware" and we can also see a trend that they mature into writing and designing software that is more testable. All of this results in better code and quality software.

Software Quality Rank consists of five improvement steps that we have attached with different meanings.

### 2.2. First Rank – Quality Awareness

In this phase, you select the code to be targeted for improvement. If you have legacy code, you must select which of these components that could be targeted for improvement. This means that you have to be "quality aware", or in other words, know what quality your entire software has. This awareness will make it possible to focus your efforts in a cost efficient way. You select targets by a series of actions. We suggest that you investigate what tools you have at your use, or can get to work in your environment. With these tools, you collect basic measurements on your code. We suggest simple measurements such as Lines of Source code (without comments), what your current test suite yields in coverage, any complexity measurements that could contribute to understanding the software, for example, call-pair, nestled calls, McCabe's Cyclomatic Complexity, Halstead Volume metric. All of these measurements are just contributing factors to support the judgement of selecting improvement targets, but should not be defining. The most important measurement is to use known failure statistics of the different components and try to establish the number of faults for each component. All the software that is in production, in customer use, and performs well should be "removed" from the target list. With performing well, we mean components that have none or very few and seldom reported failures (anomalies) that too of low severity. The target list now only contains candidates for improvement, but they should be prioritized, from the worst components (many high severity failures) to bad (some failures) in a strict ranking order. To this list you should now apply business aspects, which imply that you remove short-lived components, low-usage, low market value etc. The reason is that it is not economical to invest in these components. What remains now, is a list of the possible targets among legacy components, where your developers agree these are all the main targets of improvement. You could also add subjective aspects of selection, e.g. components with bad design,

component code difficult to maintain, written by many different developers etc. The next part of creating the targets is to prioritize all new code. The reason is that they have not been tested and we can assume that they are more fault-prone than legacy code. Depending on the amount of new code, which is often substantial, our observation is there will not be enough time to target all new code created. We suggest you to prioritise also the new code based on importance. Finally, the new code is impacting the existing code – which we call modification code, should be selected on a case by case basis. Sometimes you change very little, but in an intricate and sensitive part of the software. Even minor faults in these parts could propagate to severe failures. Sometimes the change is a minor change even if it affects a lot of code, and then selecting this area as target might not be the first choice (even if we suggest testing should not be ignored). These three "lists" of legacy, new and modified target components should now become one list, with an emphasis on new code, some really bad legacy code, and the most important modifications. If any of the lists coincide, for example, if the new code will impact the legacy in a bad place and modifications are risky – this could be a target. The resulting list should be what the developers themselves believe are the most important areas to spend additional time to improve the quality on, since if they do not believe it – it will not be done at all. This is the "baseline" of what we suggest should be targeted in the next project for special quality improvement. In a time to market software development, what *should be targeted* for improvement, will probably take too long for the projects time limits. Therefore, it is important to do a selection from the top of this list, within time and budget limitations and document that in the component test plan for the project. We suggest that the selection should also have in mind practical aspects, for example that all developers have at least one selected component each, so that the selection does not become unevenly distributed, but a team improvement.

This **component test plan** is a key document for SQR. It is important for the long and short term (project) view that defines what should be targets for quality improvements. It will work as a suggestion from development to management of what should be targeted, and if management does not agree with the limitations of the list, more time should be added to development, or fewer components could have that extra quality attention. The plan will also serve as a "contract" and will be commitment from the developers, and a good aid for management to follow up progress.

The final requirement in this quality awareness phase is to investigate the possibility to perform the next steps of improvements. This means that tools to measure coverage must exist that can handle automated test suites and a test environment that works with these tools must be created. If no tools exist, they must be procured, installed and sufficient training should be provided as a part of the component test plan.

Now the project only have one last part to define, that is to make a checklist adapted to the own tools, process and terminology. This checklist is a way to check that the selected components have reached its rank level. Each improvement requirement for each rank can be transformed into a question that will be checked before delivery. These checklists make it possible to follow up and hand-over correct information to stakeholders, such as project management, test organisations and line management.

### 2.3. Second Rank – Quality Improvement

In rank two, the actual improvement is performed on the selected targets from rank 1, as defined in the component test plan. Rank 2 will define requirements for these improvements, where the most important task is to perform then (learn how). Reviews are common practice, since many decades, in Ericsson, and every project defines which documents should be reviewed. We believe that code reviews and quality improvement reviews are often dismissed by development projects, which usually spend their effort reviewing design specifications and requirements to make sure they are clear and understood. In rank two there is a requirement that code should never be a *one -person responsibility*. It is important that more than one person have reviewed the code. We do not believe it is cost efficient to review all code with the entire design team, but it is important that selected parts of code should be reviewed. In particular, we suggest that header files should be reviewed, since they specify interfaces, parameters of importance and other valuable information. We have the requirement that input value boundaries (maximal and minimal values) should be explicitly mentioned in the header files, to ease the creation of test cases. In addition to the above focus, the review should also include if the code is effective and if there are special dependencies to other components that matters.

This is also the place to measure the percentage of comments, where 0% is also acceptable. The number of comments depends greatly on how they are written. The quality of the comments are more important than the number, which means that comments are reviewed with the aspect of maintainability and usability for another developer.

It is difficult to define the minimal documentation for a component that would enhance understanding and transferability. We suggest the best approach is to try and capture what a developer would tell a fellow developer, to add just the right information that minimizes the time to read and understand the code. We suggest that the easiest way is to use modern tools, basically have the main designer explains the code, and either make a video-clip of this information into the software repository or take a photo of the sketch and make that a part of the documentation. The least efficient way is to spend hours drawing flow-charts or creating full-fledged state-transition diagrams. What surprises us is that most people draw a sphere-arrow diagram when explaining code that could easily be transformed into a state-chart diagram, where of course some spheres are complex – and that one must either accept abstraction or go directly to view the code.

Other targets for review are the automatic test scripts, since we have noticed that this code is often at a much lower quality, containing hard-coded values instead of a maintainable test-script.

We suggest that simple measurements from the reviews should be collected at this stage. Suitable measurements could be, i.e. number of participants, time of preparation and review, and the number and severity of faults and improvements. These simple review metrics will not require much extra effort, but will give the group an indirect way to evaluate their efforts in review, and stay focused.

Assuming that the code is prepared, we have a requirement that static analysis tools should be used. In the first version of SQR, the focus was to categorize the different warnings from a Static Analysis tool (i.e. Lint, Flexlint) in five different stages. The reason being that there are a lot of warnings and it is not easy to judge the importance of the warnings. We do not want to over-exaggerate the contribution of such tools, but they are definitely helpful. The aim is to make developers see these tools as an extra pair of reviewing eyes, and thus as an aid in their task to desk-check their own code, instead of a burden of abundant set of warnings. The aim is to execute the code with the tool, analyze the result, and correct as much as possible. If warnings remain in the code, any new warnings are easy to miss. We think that the tool Coverity have brought this to a new level, finding problems rather efficiently. We are of course recognizing that there is a lot of different static analysis tools with different advantages and disadvantages, e.g. Parasoft, Lint and similar [1].

In addition to reviews and static analysis tools, the main task of this phase is to make a sincere test improvement effort.

This means that we teach testing techniques [3], such as equivalence partitioning (EP), boundary value analysis (BVA), state-transition testing (ST) and the different coverage measurements as structural techniques. The requirement is that at least equivalence testing is performed, with exercising both allowed and disallowed parameters (the latter we call “negative testing”). We have noticed that many developers in large complex systems tend to execute their load module in the existing context of the system, instead of spending their time to stub every aspect. They are also primarily using functional (traditional “black-box”) test approach, where they are exercising the normal case of the component through its interface. Here the aim is to make developers aware of the limitations of such testing, with the first goal to create many more functional test cases that would be as complete as possible from a functional point of view.

We measured the number of test cases that existed, and how many new test cases have been produced, and did not pay attention to how big a test case is. We have the requirement that normal cases and the most common fault cases should be executed. We encourage an “automatic regression suite” of the component to be created by programming test scripts. These test scripts should be treated as normal code, have header information etc. The test scripts do not need extra written documentation other than a very high-level test specification. The test script should clearly specify what it tests. There is no need to say how much or what should be automated. We assume that developers like writing code, and this is a natural way to create tests. What might be new is using a test harness tool, or a test framework with templates available, which will ease the maintainability of such test scripts.

This test suite are then measured with a coverage tool, which should give the developer adequate feedback on how well it is tested. We have used the general assumption that 50-70% statement coverage means the normal cases of the code have been covered. To test fault cases, you have to add more test cases. Using coverage to create test cases is a good help to make sure the code is understood. A lot of faults is hiding in those fault cases.

We encourage 100% feasible statement coverage [2], which gives room to decide what is feasible (economical, cost-efficient, possible) to perform. A component can sometimes consist of several hundreds of files. A deliberate priority within the component should be done on what files that should achieve 100% statement coverage and which should not. The average of the component could be as low as 85%, since good code in our context is assumed to consist of many security and safety entries that can never be reached.

Also we have noticed that software that handles hardware might sometimes be harder to test in full (as well as kernel code of operating systems). 100% statement coverage means that it is not completely tested. At the end, it is good testing we want, and not a good measurement. Yet, the developers have to be able in an assessment to justify their achieved coverage and explain any low numbers. Priority within the component is vital. We have seen the coverage to be a very beneficial tool, if used with sense, and in the order we have suggested.

Finally the rank 2 requires that memory checking is performed using tools, such as Purify. We have also an option that profiling can be used if it is applicable at this low level. Having an efficiency check of the code is useful, since many small components contribute to the overall performance. It is a danger to make this mandatory, since sub-optimizing might not be efficient, but one example is that if it is possible to measure e.g., send or receive something that might be easily timed at this low level. It is a good stage to capture performance problems. All these items are then checked, and documented with appropriate logs and references in the checklist that is delivered with the code.

### 2.4. Third Rank – Transfer Quality

The third rank is aimed to be the goal for most components (95%) in commercial software. This is the most cost-efficient quality improvement, and spending more time will find more faults, but requires more time and resource investments that needs to be justified. This rank level is based on what senior developers, with good quality sense are doing to make sure that the code and its documentation is sufficient, the code is possible to transfer, and the code is maintainable without extra investments. The idea is that the improvement here becomes only some direct actual doing, but more of a checking of the component to make sure all documents (incl. test docs) are in place. The reviews performed should be with the additional focus that the documents are good enough to "handover" to another party, and that review meeting should be with stakeholder's presence. Here, testers can be invited to review test scripts and test specifications, and a maintenance organization can participate in both design and code review for selected parts of the software. This could be planned from the beginning of the project, and rank three should not be a costly phase to achieve.

The focus is again to improve test by adding tests – by exploring the input better, i.e. making a boundary value testing (three values for each boundary). Also loops, nestled calls, implicit else etc should be explored. The aim is improve the testing with more and better fault scenarios, and the goal is to reach 80% feasible branch coverage, and explore basic conditions if they are prioritized. Parts of the code could be target for state transitions or state chart testing. Initially we had several measurements (complexity) here, but this has been dropped, since we feel they do not contribute to quality improvements. The aim is to conclude that the component has been tested, measured, reviewed and have sufficient documentation to be transferable with a small cost. In rank 2 we believe the component is ok, where as in rank 3 we are confident will perform ok. Yet it is important to point out that this is a cost efficient judgment on the component, and we have made an effort to find the "right" level.

### 2.5. Fourth Rank – Critical Code Quality

At rank 4, we change the concept from discussing components to discussing code. In particular, we are selecting critical or central parts of the code, within a component that should be of rank 3. This could also be code that should be optimized for performance, memory utilization, size or similar constraint. Here we claim that if that part of the code is so important, a complete state transition diagram should be created on that critical section of the code. This code (and its dependencies) should be a subject to a more formal inspection. In addition, better failure scenarios should be discussed to try and create the code section as fault-free as possible, and here 100% feasible branch coverage is the goal, but we suggest to look at other coverage measurements that are applicable (e.g. Linear Code Sequence and jump, that is often called "loop coverage"). We assume that by selecting a part of the code for rank four means that the appropriate additional improvement is conducted e.g. analyzing messaging sequences. Optimized code is often more difficult to maintain, which implies a better documentation is needed.

### 2.6. Fifth Rank – Safety Critical Quality

We are aware that for safety critical code, a number of standards exists that are mandatory and that provides useful guidance for developers. We are just making it clear that this is also the high-end of the quality scale, and gives a perspective for quality. The requirements are to perform formal inspections of all code, perform a FMEA-analysis, but also to use at least two different static analysis tools and two different memory tools (since they find and enhance different problems).

Profiling tools should be used if applicable, and code should be used with all strict compiler flags set.

We have also noticed that executing the code by different compilers can weed out some intricate compiler faults. If the code is safety critical, the documentation must be complete, and include training material. The coverage requirements are at least 100% state transition (n-2) coverage and 100% MCDC coverage. In addition we suggest applying the domain standards, e.g. DO-178B, FDA and IEC 61508.

Unfortunately, we have not had the opportunity to explore the rank 5 improvement within Ericsson yet, but some of our external users (medical and defense) explained that the SQR scheme has been valuable for the non-critical code, to make a more controlled distinction of the quality levels of the code.

## 3. Experiment

Our experiment was within one world wide project on a product with distributed design teams consisting of 8 sites/organization and 22 design teams. These design teams were more or less in parallel, and all organizations had different history, motivation and attitude to this quality improvement. We are trying to describe these teams in a fashion that could be useful for others with the aim to deploy SQR. We realize that also culture has a large role, where e.g. Swedish developers need to be convinced on a more personal level than more eastern cultures. We have though concluded that developers favor the scheme when they understand the time-negotiating principle of quality, and that the aim is really to make the work developers spend on quality enhancement more explicit for management.

There is a strong tendency that the word of mouth – success of others, is the best motivator. We initially spent more time with people who were willing to use the scheme – and were more quality aware from the start, which made it easier to sell the concept to others if they had success and approved it. Therefore, our target persons were the senior developers in the teams, that would probably do most of the suggested work any how, and the effort would not seem so insurmountable. The senior designers are informal leaders, and they took the initiative to put tools and environment in place for the rest of the team.

### 3.1. Organization A

This organization did only perform SQR, and kept the process as is (see the discussion in Validation section). Therefore, this organization is the one of the few that had a quality improvement based only on SQR. This organization had two design teams. The first was known to have better quality from the start than anyone else, and could be viewed as quality aware. They particularly appreciated to move from only functional testing approach to a more structural approach, and were welcoming tools, guidance of what input to select, test techniques, and how to best utilize code coverage. This team quickly selected one person to do the main coverage improvement, but many of designers improved their code according to the concept anyhow. Rank 1 was not performed, so scope was the entire software. This resulted in a doubling of resources, where almost all parts targeted reached rank 3. Here reviews were already a part of the work and test automation mandatory. The long term result resulted in a flawless code, and very few faults were found during the next two test phases.

The second design team had responsibility for new hardware, and most of the personnel were new to design this type of code. Also, some of the code was outsourced, which added more risk. This team was humble enough to ask for a lot of help during the process, which we believe is one of the contributing factors. They were also open for external assessments, which had a positive effect on quality – If you know someone is going to review your results, you put more effort in. At the end, they had trouble achieving the coverage measurements, mostly because of tools problems, and the lack of suitability to do coverage on kernel registers with available tools, but a targeted quality effort brought the measurements up to sufficient levels before release (which was postponed). In particular, we assessed that the conscious review and test targeting were the most beneficial parts of this team's success, since we believe review helped the new team to understand the context of the product. The conclusion was that they reached rank 2 for 60% of target and 10 % rank 3. The rest not fulfilled rank 2 in especially for the coverage part, but in most other aspects. This team when delivered to the next phase, functional test, saved five weeks out of the normal six that was the previous average for this hardware test, which made the testers to be moved to other teams, since quality criteria was already fulfilled. The remaining work was with the outsourced part that had not fulfilled the quality requirement and had difficulties in testing their own software in their environment.

The initial cost was expensive in this team (more than double the cost in time of design, but the savings of these two teams were so obvious, in all later phases, that this impacted the entire project.

### 3.2. Organization B

This organization was early adopters of both the new process and the SQR concept and consisting of 4 design teams. They were early selecting strong

champions to create an adopted checklist, that later became standard within the entire project, and introduced tools with tool champions, such as Test Real-time (from IBM/Rational). Much of the test scripts were already written in an internal tool based on tcl, and these scripts were possible to measure using the Test Real-time tools. These teams did an initial assessment to understand their current status and attitude and SQR started a strong internal debate on quality. They took on a too big scope, mandating all new and changed code should reach rank 2, which was followed more or less enthusiastically. This made the internal assessments and follow-up a bit too loose, and the request to provide logs on actual coverage came at a late stage. Three of the four teams focused on increasing the number of tests. Static analysis was for one of these teams considered a good contributing factor spotting 18 real faults in the first run. Reviews were made by all the teams, but again coverage was late to be used as a tool and the test somewhat different for all but one team. This was the team that had the test tool champion, that delivered full automation and good coverage, with many new tests added, but a long time was spent on discussion on how coverage was actually measured. Teams with strong champions had better results. The team with the poorest initial result was the team that delivered code generated from RoseRT. It was a problem how coverage should be judged, since a lot of generated code is unreachable. This team did then really make an effort, and improved there results substantially. Within this organization there were only one team that had a low and a high demand for quality, and these were late adopters within the team. Here a result was also that this team transferred its code to another organization.

The conclusive results for these teams were that about 50% reached 90% of the rank 2 requirements and 40 % reach approximate 70% of Rank 2 requirements. Only 10% of the ranking achieved rank 2. We believe reason was the wide scope was taken, and rank 1 selection was not targeted enough. Nevertheless, this was considered a substantial improvement, with many strong champions still working within the teams. The main problem we observed was that during the next project the quality approach was lost. We believe that the management did the wrong judgment that *"now when the code has reached its quality - we can cut for development again*." This was a mistake, that even if all touched code was a target for quality improvement, it was not completely fulfilled, and never reached rank 3, and also a new project will target completely different parts of the code, that modifications and the added new code will still need its targeted effort. We also discourage the fact that all new code must have Rank 2 which we believe is an impossible task with the time given and will work as a discouraging factor that makes the checklist fill in an additional administrative effort instead of a targeted improvement.

### 3.3. Organization C

This organization consists of 10 different design teams, whereof 8 of 10 did attempt SQR and two teams "cheated" by filling in fictive values. This became revealed when we reviewed failure reports, where all teams have improved substantially except these two. We guess that no one would believe that someone seriously would cheat, and rather bought the talk of "these components being so special". This organization had weak initial interest, and made a very minimalist checklist, which was later abandoned for organization B's checklist. The first two teams got no extra time, and not until organization A and B had started to show good results, this organization took a serious look. In addition, the initial champion had moved away to a new role, and the managers were supposed to drive the improvement, something that failed in all aspects. The top project management had to assure that time was really given to achieve the requirements of SQR 2, and the contract principle of the component test plan won developers. Then the team started to catch on, by creating test environments, using test tools, and targeting the right components. Here the "second best" persons were getting real results and could actually see how they were saving time for themselves, which finally convinced many people to change. This team did also have a lot of rank 4 components, but the lack of sufficient tools, substantial stubbing, etc impacted many teams to get the real success. It is still hard to judge the result in factual numbers on coverage in these teams, since they were sensitive to outside assessment, but the test maturity has improved tremendously for this organization – which is the most important result. Not only did the quality for 8 of 10 teams get lowered to 10% of their earlier average, but they also introduced several steps of testing within the development phase. So even if the new process is used, they moved during this project from one test level to wanting four internal test phases before release outside their organization. The four levels are designer (stubbed), component, multi-component and functional test. The conclusion is impressive, and the remaining faults are often so intricate that they are hard to trace and debug.

### 3.4. Organization D

These were the earliest adopters of the concept, being suppliers in an external organization they saw this as a requirement, and were the first to do the checklist. In one sense, these teams were the most experimental to the concept. In practice, we assessed that they did well in all aspects of SQR that they personally believed. They never understood contracting principle internally in the component test plan, but we believe they targeted software, even if only a limited extra time was added. The main reason is that the benefit of improved quality would result in less work for the design team, since the savings would be at test levels at later stages and outside the development organization. At this time, that was not a positive factor, yet we could see a will to make this happen. The review concept, as a quality contribution, was never taken seriously, and treated as a hand over between two developers signing off the code. In all other aspects, we believe the SQR was followed. What was impressive in this team was the management engagement, and that aim to perform well, where some energy was set on test. They claimed themselves having great success with the scheme, but it has been hard to review from the outside. The result of this code was in large parts outsourced further, due to economic pressures. The most interesting result in addition to the quality improvement (where all selected targets did reach rank 2 according to them), is that the remaining 75% of the faults were related to memory problems for this area, and that memory tools were used at the lowest level, indicating that memory problems can remain in later stages of testing.

### 3.5. Organization E

This team was a very tight team with partly "unreasonable" quality demands (all or nothing) approach to their software. They adopted the new process and SQR at the same time, and what was particularly interesting is their approach to static analysis that found many faults. An intense use of the tool, with dedicated two days with the entire team and one champion that had learned and been champion of the tool. Coverage figures were debated, but again understanding the selection was not targeted enough, and became more on the individual designers time and interested for its component. In many aspects, they did not fulfill rank 2 at all. In later analysis it became clear that their budget was structured on maintenance, and a too good quality would have lowered their budget (and giving them less work), which is not a real internal incentive of becoming too good for the organization.

### 3.6. Organization F

Organization F was consisting of two teams. This organization could easily recruit people, and many designers and testers were relatively new working in Ericsson. Good management and early teaching on testing made this team very interested in this quality improvement, and they were very happy that there was an increasing quality demand on the product, as they saw as an opportunity for them. These teams embraced the concept of SQR and were listening carefully to any testing advice. The most problematic issues were the component test plan and the contracting principle, which they felt new in their role and it was difficult to make a case with their management. Yet their focus and interest motivated them to have good test behavior early, and extra persons were added to the team instead of giving them more time. Automating test was a conscious decision of the entire team, which also proved valuable. The result showed that rank 2 was achieved in most cases, and the quality was substantially improved. No later follow-up has to be conduced, but we know for a fact this spread further within their organization.

### 3.7. Organization G

Organization G was a mid-size team and has no introduction or training in SQR ideas, except what was written. They did instead make their own "unauthorized" checklist that was a simplification. In later review of the checklist, we found many principles that were interpreted in a questionable matter. This team had not understood the word "feasible" coverage, and delivered 100% statement coverage of all their components. Yet, no other functional testing was made, which resulted in fault-prone software that had many integration problems. The remedy was sending several seniors on place to try and educate the team. There were no particular SQR assessment or follow-up in this team, but we mention it to point as an example of how easy it is to misunderstand and misuse a good concept, achieving "on the paper" some metrics, and yet failing the quality.

### 3.8. Organization H

This final small team of 6 persons and one tester, but medium size software, was an enigma. In the initial teaching of SQR, we believed they had not grasped it, but on follow-up, they exceeded all our expectations. In later reflection this should have been judged as winners, since they at their first meeting could present factual information on their actual quality, which is a good indication of control.

They grasped the internal (somewhat secret) ambition of the entire SQR project that at least half of the number of current faults should disappear after this improvement. The thinking was if all individual designers improved, the overall sum of faults would be substantially less. This team had only 56 failures on their software (and only acknowledged 26 of them as true software faults). The goal was set at 13. There is not much to say except that they followed SQR with their own checklist in all aspects, and the final result was only 6 faults were found in later stages, but also this team managed a much earlier delivery and the code was considered as a high quality code. The team was definitely a very quality aware team, where all components selected reached rank 3, and 2 according to plan and it was hard to find anything that could have been done better given the limited time and resource.

## 4. Achievements

In conclusion the failures in system test dropped to 10% of the earlier versions, and the conclusion is that the SQR concept saved 67% technical hours, diminishing the time for maintenance substantially. We could see a reduced failure administration from hundreds of problems a week to 2-3 failures every second week.

The savings came in all later phases of the project, which are different levels of testing. The assumption is if the quality is good (great) from design, all sub-sequential test levels, including corrections of code, administration of failures etc will save a lot of time. The reason is that a quality product will be faster and easier to install, test suites will execute faster and less regressions and re-deliveries have to be made, all factors that save time. The quality improvement during this 18 months project actually challenged the system testers to re-design their test cases, and left room to handle a lot of change requests, get control of back-logs and basically return to a more reasonable working situation. It is no secret this product was pushed a bit too hard in the time to market race for release, and a bit too many quality problems were a result of cutting too many corners in earlier releases. Now this product is viewed as the best in class, when it comes to quality compared to its competitors. We definitely think that the new process and SQR together have unquestionably moved fault finding to the earlier phases in development life-cycle. It is easy to view the result, where we could see that now the organisation (A-H) finds their own failures to 85% and the next (internal) customer and external customer only finds 15% of the failures. The figures before this improvements were vice versa.

## 5. Validation

The most problematic validation of the SQR improvement is that a major process change was introduced at the same time in this project. This moved the process to a more integration centric development, where developers and testers worked together in teams. The process improvement was appreciated, and solved some of the difficulties with large complex software, such as the problems of incompatible interfaces and instead focuses on frequent builds and constant integration. Unfortunately, this makes it also difficult to give all credit to the SQR scheme. Therefore two additional investigations were conducted, one conducted as a master thesis [4], which reviewed the scheme and interviewed 30 participants. The conclusion was that all but one person agreed that SQR was strongly contributing to the quality achieved rather than the process. All agreed that SQR put the focus on having sufficient test tools and importance of good test environment. The interviewed persons also had to select the most contributing part of SQR, where coverage measurement was most popular, followed by the static analysis and code review. The part of the contribution was getting direct and fast feedback on how good the test suite was.

In addition to this, a second independent investigation was done where all managers voted (A-H) in this 1200 person large organisation on what had been most contributing quality improvement. First place took the new process improvement, but software quality rank took second place, and “improved component test” took third place. We are of course puzzled in that distinction made, but nevertheless the contributing factor was an awareness that is undisputable in money saved, time saved and quality achieved. It is no surprise the SQR is deployed in many different ways across Ericsson.

The validity of these results can be debated. We believe such a substantial improvement can not be achieved by mere focus on quality or “Hawthorne effect”, so these practices suggested must have quality implications. We feel the statistics collected have so many flaws in the way it was collected and reported not to mention the discovery of teams misusing the reporting, that we are very hesitant to make statistical treatment of the data. We prefer to handle data as trends in the context we have described. At the end of the day more controlled experiments should be done, and it is only the scale of this that gives an indication that the results have some substance to it. We encourage others to collect better measurements for scientific and research conclusions.

## 6. Discussion and Insights

The most common question we got during the introduction of this improvement *is: What is a component?* Our explanations have changed, where we have ranged from: managed item in the Configuration Management tool, a conceptual item, the smallest executable (identifiable) piece of code, to a more general item with a clear interface that could be executed in isolation. We view the debate as a decoy, and this will be settled when starting to work with the tools. If the component is set on a too high level, the coverage result is too difficult to achieve.

The second most common question is what coverage is enough. Again, all code is not equal, which is an important factor. Also, it is the designer's confidence and aim that is important. 100% feasible coverage (assumed statement) does not mean tested. This is clearly shown when discussing conditions, loops, and input, and developers becomes aware of this fact. Therefore, we allow differences of coverage within the component. The follow up question is what coverage should be demanded? We believe our approach to feasible testing has a strong case, but think that the question reveals the wrong attitude. It should instead be: *How do you know if you tested enough?*

We also believe that there exists no real answer on how effective reviews are. It is difficult in practice and depends on how you perform the reviews.

Static Analysis tools can be debated [1], but we review the use of this is more common with quality producing developers.

Our final question we would like to answer is, why all do no have same success in quality when adopting SQR? We assume activities were not preceded by an assessment that showed component test as the main problem. Secondly, we believe that often the initial Rank 1, making a targeted selection was ignored, and a too wide scope was selected. Scope, ignoring the contracting principle, and a lack of motivation, seems to be the main reasons to fail with the SQR scheme.

## 7. Related work

Many solid research papers have been written in this area, encompassing review, static analysis and coverage. Unit, component or module testing is not new either. There is no secret that we are inspired by Software Engineering Institute's Capability Maturity Model (CMM) [7] in the way to stage this as 5 levels. The main difference is that CMM addressed the whole process whereas our focus is on component testing alone. The Swedish school-system have had a 5 level grading, where 3 means "pass", which made us – in contrast to CMM, not thinking the ultimate goal is that all code achieve level 5, but that the majority of the code should instead should achieve "pass".

## 8. Conclusions & Future Work

Ericsson did not only cut delivery time to less than half, but the quality improvement was substantial. We believe strongly that stepwise improvement is the best approach, but there should not be too many steps to achieve, as this adds complexity. We also conclude that component test is probably a focus for many organisations with time to market software, in addition to system testing. We believe the focus is the developers that at least within Ericsson have a major quality impact. Therefore it is crucial to provide developers with better tools and test environment. Our future work will be to assess how this method changes and is applied when the originators are not at hand – and to continually assess the results of these changes. We have understood the difficulty of proving these experiences in a purely scientific setting.

## 9. Acknowledgements

Acknowledgement goes to the developers that have made the improvement, and the Ericsson Management supporting this work. This paper was funded by the Swedish Knowledge Foundation SAVE-IT program, through Department of Computer Science and Electronics at the Mälardalen University in cooperation with Ericsson.

## 10. References

[1]Emanuelsson, P., Nilsson, U.: *A Comparative Study of Industrial Static Analysis Tools*, Linköping 2007, to appear.

[2] Zhu, H., Hall, P. A., and May, J. H.*: Software unit test coverage and adequacy*. ACM Comput. Surv. 29, 4 (Dec. 1997)

[3] Reid, S.,: *An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing, metrics*, Fourth Int. Software Metrics Symposium (METRICS'97), (1997)

[4] Stenmark, J., Boqvist, H.: *Analysis and evaluation of Software Quality Rank performed on Component Test*, Master Thesis, MDH 2004

[5] Eldh, S.: *Software Quality Rank – An Improvement in Component Test***,** Proceedings of the International Conference ICSTest, Dusseldorf, Germany, April 2004

[6] Eldh, S.: *Software Quality Rank- Improving Designers Test*, Tutorial of the 11th Int. Conference EuroStar, Köln, Germany, November 2004

[7] Humphrey, W., *Managing the Software Process*, Addison Wesley Professional, MA 1989