Improving the Validation Process for a Better Field Quality in a Product Line Architecture

Christof Ebert
Alcatel, Switching and Routing Division, Antwerp

Abstract: Telecommunication switching software challenges the development process with the need to continuously update a product line architecture with new features, while simultaneously synchronizing across related product lines that are operational in a global setting. Serving an installed base of over 130 million lines with market-specific variants that have been continuously growing for more than a decade needs careful harmonization of product line evolution with serving customer needs. The focus of this case study is on quality improvement with code inspections and integration testing in a product line architecture. Improving customer perceived quality and at the same time lowering the cost of non-quality are the two primary objectives of the process reengineering activity. The return on investment of described improvements was significant. Several results on the relationships between validation processes, fault detection effectiveness and product quality are discussed. Experiences from over 50 projects of various size that were developed and deployed during '95...'98 by Alcatel are included to show practical impacts that can also be transferred to other projects.

Keywords: CMM, fault detection, code inspection, integration testing, return on investment, maintenance, process improvement, product engineering, software best practices, software quality

1. Introduction

With a growing share of software cost in almost all state-of-the-art technical products such as cars or telecommunication infrastructure, it is of primary concern to control and reduce software development cost. To better manage development cost of globally available products, concepts of product lines are increasingly applied. Unfortunately, software engineering technology has been developed mainly for creating one product at a time and existing process models do not well address product line development.

Although experiences with development and maintenance best practices had been collected for many years, it is still not obvious, which technique to select and how to assess impacts. We will in this article summarize several best practices for validation in a product line architecture, which achieved improved predictability, and quality in software projects of various sizes. The challenges we were faced and which we will address involve:

- to support of validation in a global product line concept;

- to facilitate early fault detection with an improved validation process;
- to reduce overall cost of non-quality.

We will address approaches to improve the quality level in the field and in parallel reduce the cost of non-quality during development. Cost of non-quality is the cost of not reaching the desired quality level at the first run. It is often referred to as "rework". We calculate cost of non-quality by summarizing respective (phase-depending) cost for fault detection and correction across all faults found in the project.

Our focus is on early fault detection. Obviously the process reengineering program needed to integrate well with ongoing product line evolution as new features are continuously integrated in such a system.

Three hypotheses had been set up before that study and were driving the described part of the improvement program as such:

Hypothesis 1: There is an optimum code reading speed that balances the conflicting requirements for high faults detection rates and low fault detection costs;

Hypothesis 2: Increasing the share of code inspections would improve design fault detection effectiveness and thus reduce normalized cost of non-quality and customer detected faults;

Hypothesis 3: Increasing test fault detection effectiveness (i.e. percentage of remaining faults being detected during test before handover to customer) would reduce normalized cost of non-quality and customer detected faults.

We will analyze these hypotheses in the context of over 50 projects developed in a three year timeframe. Mentioned hypotheses also reflect the type of changes that we focus on in this case study. This study is unique in that it provides insight in a running software process improvement (SPI) program within a large organization dealing with legacy software.

There are few documented results around that describe improvement programs together with quantitative data. Most results published so far describe the background of such a program with focus on the assessment and qualitative observations [3,4,5]. They are in many cases looking on rather small groups of engineers that act like a small- to medium-size company, even when embedded in a big organization. The recently published results of SPI initiatives within CMM L5 ranked Boeing Defense and Space Group [4] and a group within Motorola [6] surely help understanding the value of moving the long way for CMM L5. Many qualitative lessons learned have been documented, but they are difficult to scale up towards large legacy based development projects. Often they seem not to be related to quantitatively specified upfront expectations.

Several current best practices related to configuration management, inspections and validations within a product line concept involving legacy software are elaborated in [9,10]. Best practices on software metrics and technical controlling are described in [11].

Few recent studies investigate the added value of a SPI program from a quantitative perspective [1,2,7]. They try to set up a return on investment (ROI) calculation that however typically takes average values across organizations and would not show the lessons learned in sufficient depth within one organization. It has been shown in these studies that the SEI CMM is an effective roadmap for achieving cost-effective solutions.

Within this paper several abbreviations are used that might not be widely used. CMM is the capability maturity model; SPI is software process improvement; PY is person years and Ph is person hours, r is the correlation coefficient measured in the observed data set. Size is measured in KStmt, which are thousand delivered executable statements of code (incl. declarations). We prefer statement counting compared to lines because the contents of a program are described in statements and should not depend on the editorial style. Failures are deviations from a specified functional behavior. They are caused by faults.

The paper is organized as follows. Chapter 2 provides an overview of the product line and development concepts used. Chapter 3 summarizes the set-up of the study. Chapter 4 summarizes the improvement approach related to validation in a product line architecture. Chapter 5 deals with quantitative validation of mentioned three hypotheses related to achieving better product quality and reducing cost of non-quality. Chapter 6 summarizes effects of ROI calculation. Finally, chapter 7 summarizes the results and raises some further questions on how improvement activities might evolve over time in a product line organization.

2. Study Setting and Product Line Concepts

The *Alcatel 1000 S12* is a digital switching system that is currently used in over 40 countries world wide with over 130 million installed lines. It provides a wide range of functionality (small local exchanges, transit exchanges, international exchanges, network service centers, or intelligent networks) and scalability (from small remote exchanges to large local exchanges). Its typical size is over 2.5 million source statements of which a big portion is customized for local network operators. The code used for *S12* is realized in Assembler, C and CHILL. Recently object-oriented paradigms supported by C++ and Java are increasingly used for new components. Within this study we focus on modules coded in the CHILL programming language, although the concepts apply equally for other languages. CHILL is a Pascal-like language with dedicated elements to better describe telecommunication systems.

In terms of functionality, *S12* covers almost all areas of software and computer engineering. This includes operating systems, database management and distributed real-time software.

The organization responsible for development and integration is registered for the ISO 9001 standard. Development staff is distributed over the whole world in over 20 development centers with the majority in Europe and US. Projects within the product line are developed typically within only few centers with a focus on high collocation of engineers involved in one project and a high allocation degree to one specific project at a time. In terms of effort or cost, the share of software is increasing continuously and is currently in the range of 80 %.

The product line concept is based on few core releases that are further customized according to specific market requirements around the world. The structuring of a system into product families allows the sharing of design effort within a product family and as such, counters the impact of ever growing complexity. This makes it possible to better sustain the rate of product evolution and introduction to new markets. There is a

clear trade-off between coherent development of functionality versus the various specific features of that functionality in different countries. Not only standards are different (e.g. Signaling or ISDN in the USA vs. Europe) but also the implementation of dedicated functionality (e.g. supplementary services or test routines) and finally the user interfaces (e.g. screen layout, charging records).

Product line concepts start with extracting commonalties in these functions and trying to separate in architecture and development from what can be considered as customization. This split however is difficult and often not supported by architecture as the needs gradually involved with new markets and growing globalization. Several product lines might co-exist for distinct core functionality and the development must be carefully split to avoid redundancy. Gradually such lines are merged, for instance with the introduction of new underlying technology.

The key roles we distinguish in the product line development are as follows:

- The core competence team of highly experienced developers deciding on the architecture evolution, specifying features, and reviewing critical design decisions in the entire product line.
- Developers responsible for designing and integrating new functionality for all software including database population tools. This involves detailed design, coding, inspections, module test, and unit testing until the functionality is integrated.
- Testers who maintain a continuous build for each single project.

With reduced scattering of engineers across projects, there is not anymore the global owner of a specific file across projects. Instead many developers in different places simultaneously share the responsibility of enhancing functionality within one product. Often a distinct file is replicated as variants that are concurrently updated and frequently synchronized to allow the centralized and global evolution of distinct functionality.

The improvement program within S12 development to which we frequently refer consumes roughly 5% of the total development effort for activities such as process control, pilots, training, tools improvements or enhanced tracking activities. It is based on

- Setting yearly business objectives with respect to processes and product line evolution;
- Investigating the current status with respect to these objectives with means of checkpointing;
- Introducing process changes which are closely related to the project roadmap of the product lines;
- Monitoring the effect of process change with in process quality checks and technical controlling.

For this study we are providing data gathered during development and field operation of several releases of the *Alcatel 1000 S12* switching system. Over 50 projects are investigated that had been developed over a timeframe of four years. These projects adequately represent the entire product line and were selected based on their business impact. Only very small projects are left out. A project as we refer to in this study is the customization of existing functionality for a distinct market, or the development of major new functionality within one product line for a lead market. They typically comprise between 10 and 200 PY.

Project size in terms of effort ranges between 10 and 500 person years with around four million new or changed statements in total. Each of those projects was built on around 2500 KStmt legacy code depending on the degree of reused functionality and underlying base release. The field performance metrics used in this study cover altogether several thousand years of execution time.

The data for this study was collected between '95 and '98. The first projects started before the improvement program was implemented, thus helping in defining a baseline of field performance without implementing process changes. Metrics are collected automatically (e.g. size) and mostly manually (e.g. faults, elapse time, effort). They are typically first stored in operational databases related to the respective development process, and later extracted and aggregated for the project history database. Since the projects are typically developed in a timeframe of several months and then deployed to the field again over a period of several months, the timestamp for each project in time series was the handover date. This is the contractually fixed date when the first product is delivered to the customer for acceptance testing. Customer detected faults which is one of the dependent variables are counted from that date onwards.

Fault accounting is per activity to allow for overlapping development activities and incremental development. This means that the major activities of the entire development process are taken for reporting of both faults and effort, even if they overlap with a follow-on activity. If for instance a correction was inspected before regression testing, the faults found during that code inspection would be reported a inspection faults, rather than test faults.

3. Research Methods

We are using an observational field study for the empirical investigation. Three hypotheses had been set up before that study and were driving the described part of the improvement program as such:

It is a field study because data is collected from different projects simultaneously. The development projects were conducted regardless of the needs to collect the experimental data. The data was mandatory reported by the project staff during the respective development processes. Similar to a case study, a lot of detailed data is collected per project. In total several hundred measurement points across processes and over time are available in the projects' history database for each single project.

We describe this study as observational because we monitor certain attributes related to a research goal in various projects over time. The data is collected from a distinct class of similar projects within one product line during development and field operation. Only projects that followed the entire development life cycle were considered. No project was left out due to specific data values.

Having such a class of closely defined project characteristics helps in overcoming one weakness of the observational case study, namely the restriction to later replicate the projects with some variables slightly changed in a controlled way. Replication of settings can be checked by looking at dependent variables of similar projects in terms of the independent variables. A theory is built after the first few observations which is then validated in this study. The approach is similar to methods used in social sciences

and allows generalization after a sufficient number of tests have been done.

Although the projects are in a real setting with no artificial controls being used, we could control several variables during the projects. This happened with the described improvement program that focused on a limited set of improvement objectives related to fault detection. Final authority on the independent variables was naturally with the project manager, however he was limited in the degree of freedom by imposed quantitative improvement objectives he had to follow in a specific year.

The entire SPI program is instrumented with numerous product and process metrics. Product metrics are primarily needed for project tracking, while several process metrics had been introduced to gain insight in process behavior, collect baseline data for further process changes, and to find out about any improvements related to controlled process changes, such as pilots. Besides basic characterizing metrics, such as size, effort, or handover date, we will mostly use normalized metrics that either express a share (e.g. fault share found during design) or a ratio (e.g. faults per new or changed KStmt). These metrics are in the same order of magnitude thus avoiding disturbances during statistical analyses. All metrics but one are at least on an interval scale. As not all metrics can be assumed to follow a normal distribution we apply especially for correlations non-parametric statistical techniques.

All values are averages on a per project base to avoid statistical interference that can occur in too small samples. For instance if we had measured some aspects on a per module base, it would be impossible to later relate this to the other factors being analyzed that only relate to the overall project. This approach also eliminates the influence of individuals to the analysis. The more developers and testers are involved in a single project, the less the individual quality levels impact the results. For exactly this reason very small projects were eliminates from the study thus leaving 50 projects from the overall set of projects developed in this product line in the given timeframe.

Obviously many relationships of data can be exploited with statistical and data mining approaches. The focus of this study however is not to detect new relationships, but rather to follow up three key decisions we made early during the SPI program in order to improve cost of non-quality and field performance. Our contribution to ongoing research is that we validate several key relationships in real settings with projects based on legacy software. Our contribution for practical development projects is that we propose meaningful and valid approaches to achieve quantitative field performance improvement. For space reasons we will only in the conclusions of this study come back to another major improvement that just follows naturally the suggested approaches, namely a much better delivery accuracy in terms of handover date.

4. Practical Improvement Concepts

Improving customer perceived quality can often be broken down to one overall improvement target, namely to dramatically reduce customer detected faults. Delivery of a switching system typically consists of several steps. The first and most relevant milestone is the handover to the customer who in most cases would start his acceptance test. One key measurable target we introduced for all projects was to halve the normalized amount of faults found after handover to the customer every year. We took

the normalized amount of faults in terms of faults per new or changed statements to allow for comparison across projects and markets.

Reducing total cost of non-quality is driven by the fault detection distribution. Assuming a distinct cost for fault detection and repair (which includes regression testing, production, etc.) per detection activity, allows calculating the total cost of non-quality for a distinct project. Faults are accounted per detection activity to don't confuse with incremental development. The second key measurable target thus was to halve the relative cost per fault after two years.

While code inspections could be triggered with almost cookbook style approaches, the situation in test was less obvious. It became even worse with finding more faults upfront, because the effectiveness of test decreased and still many faults could only be detected in the subsystem or even within the entire switch due to the many data-driven interactions with other components. The related measurable improvement target for testing was to find more than 90% of all remaining faults in the software before handover to the customer.

Unlike the two key improvement targets that are dependent variables which relate to the hypotheses we mentioned earlier, test effectiveness is an independent variable that is directly controlled by means of dynamic test case selection and applying reliability growth models to predict remaining faults.

4.1 Implementing Improvements

Our approach to achieve these improvement objectives was as follows:

Step 1: Several "root causes" of late fault detection were investigated. Systematically we interviewed developers and testers on reasons why faults were not detected earlier. Several reasons were identified. Inspections were typically not following the defined process, involving checkers, inspection leader and a maximum reading speed. Many inspections were considered finished when the respective milestone date appeared, instead of applying reasonable exit criteria, before continuing the fault detection with the next and more expensive activity. Test was conducted with a rather static set of test cases that was not dynamically filtered and adjusted to reliability growth models. Module test typically didn't rank high in these lists which explains why we left it out from the discussions here.

Step 2: For both areas we started a distinct improvement activity with the target to define the underlying process, pilot and provide tools support, train the people involved with these activities and institutionalize the processes. For the inspection process we started a training and certification program of inspection leaders. Each development area was obliged to train a distinct amount of the developers as inspection leaders. Each project has to plan inspections with certain criteria, such as having a certified inspection leader on board, or following a certain checking speed. Checklists were directly linked to work products of the different development activities, for instance depending on programming language used, or depending on the respective subsystem with its own peculiarities. Checklists were continuously updated with the results of root cause analysis of each single field failure. Finally a web-based tool was developed to allow inspection planning even involving remote developers.

The test process was improved by means of providing better traceability from

changes and critical components to dedicated test cases. Test cases are classified already during the initial design activities in a global test strategy. This test strategy looks upon feature usage, criticality of features, and previous critical failures in the respective market. Asking in each project to automate a specific percentage of test cases strengthened automatic test and thus improved cost of test. Dynamic test filtering was introduced based on the amount of failing test cases and the feedback from reliability growth models. Sandwich test was introduced to allow earlier start of basic regression test activities that are growing by means of a continuous build towards each incremental addition of new functionality. This allows a much longer stabilization phase, however in parallel to development. The result is much better test effectiveness.

Step 3: Exit criteria were defined and applied within projects which provided clear and measurable guidelines what to achieve in inspections and test before considering these activities finished. Exit criteria involved efficiency measures based on effort per fault detected, remaining faults calculation that could be compared with cost to complete, or coverage rules for critical components.

The single most relevant exit criteria is the gate from development towards test activities. At this point, overall project cost and cycle time is influenced most in the entire project. Poor development quality immediately affects overall project performance. Strong build management must refuse unacceptable low quality of components to be integrated. Similar high entry criteria are used before the developer passes her code to the integration within the development team. To sum up, whenever a work product is passed to a broader community, the quality level must be proven.

Step 4: Dedicated process metrics were installed for the duration of the improvement activities, i.e. until the process changes could be considered institutionalized. Such metrics help the individual engineer and involved team to immediately get feedback on the process performance with respect to the project and process improvement objectives.

Step 5: To avoid the well-known trap of any SPI initiative, namely to ask for unachievable goals, we combined these changed targets with planning and estimation changes. Inspections for instance needed a high peak load of developers during the short inspection timeframe. All this additional effort and sometimes lead time had to be committed by respective middle management and team leaders to facilitate the changes.

4.2 Implementing the Product-Line Concept

The move towards a product-line oriented development impacted also the organizational layout. We increased the responsibilities of the projects' allocated resources to achieve the objectives of the projects, while the line responsibilities were reduced towards the needs of overall technical evolution. Functional departments were combined to a pool of developers which are allocated to a project with full responsibility from detailed design until the respective functionality if successfully integrated into the build. This clear split also facilitated that developers cared more for the quality they shipped because the organizational boundaries were reduced.

The product line concept implies that feature roadmaps and deliveries of both new and changed (or corrected) functionality must be aligned and synchronized. Synchro-

nization of deliveries however adds complexity to the development process. For instance, a product-line architecture asks for simultaneous correction of detected critical faults, while new functionality with different local flavors had been already added.

The difficulty starts with parallel development of projects in the same product line. Less overlapping development of the baseline and market-specific releases means less overlapping synchronization needs, but also loss of time to market. We therefore simulate dependencies within the product line in terms of accumulated cost of non-quality versus cost of late delivery to customers. The simulation takes into account the remaining faults at each point in time, and the assumed overlapping fault detection for the parallel projects dependent on the degree of overlap and the respective detection activity. All faults at a given time to be found in parallel development can be linked to cost based on individual fault detection cost. Delayed market introduction is also linked to cost. Obviously the delay is determined by start date of coding and the amount of faults to be found that are inherited from the "parent". Fault detection is not only cost but also lead time. The trade-off allows managing a product line roadmap.

To facilitate easier communication of appropriate corrections, we introduced a new synchronization mechanism into the worldwide fault database. Based on the detected failure and the originating fault, a list of files in different projects is pre-populated by telling which other variants of a given file need to be corrected. Although this is rather simple with a parent and variant tree on the macroscopic level, due to localized small changes on the procedure and data level, careful manual analysis is requested. Those variants (i.e. within customization projects) are then automatically triggered. Depending on a trade-off analysis of failure risk and stability impacts the developer responsible for the specific customization would correct these faults.

This approach immediately helped to focus on major field problems and ensure that they would be avoided in other markets - if applicable. It however also showed the cost of the applied product line roadmap. Too many variants even if maintained by groups of highly skilled engineers, would add too many overheads - that before was typically not accounted as such. Obviously, variants needed to be aligned to allow for better synchronization of contents (both new functionality and corrections), while still preserving the desired specific functional flavors necessary in a specific market.

Based on a mapping of customer requirements to architectural units (i.e. modules, databases, subsystems, production tools), we achieved a clustering of activities that allowed for splitting activities in three parts:

1. Small independent architectural units that could be fairly well separated and left out from customization;
2. Big chunks that would be impacted in any project and thus need a global focus to facilitate simple customization (e.g. different signaling types can be captured with generic protocol descriptions and translation mechanisms);
3. Specific market- or customer-specific functional clusters that would be defined based on the requirements analysis and ultimately form the project team responsible for a customer project.

Processes for project management, design, validation and configuration management are tailored within these three groups [12].

5. Results

5.1 Relationships of Process Metrics

Significance levels for all correlations that will further be exploited were far below 0.001. The highest random correlation coefficient that we generated in 1000 trials with random metric generation based on the given set of metrics observations and their distribution was $r = 0.41$. This random correlation analysis included all collected metrics, even those not used for the study described. The range for a smaller subset would thus be lower. This means that correlation coefficients higher than this limit are meaningful because even many trials with random, but fitting data, would not generate higher correlation. The ρ value based on the given Spearman rank coefficient $r = 0.5$ of the given sample of 50 projects is in the interval of [0.27;0.68]. For $r = 0.6$ the interval is [0.39;0.75].

The factor analysis started with all projects and related metrics. Commonalties of the individual metrics were estimated based on the highest correlation coefficient. Factor analysis was iterated to achieve a 1% difference between estimated (input) commonalties and the calculated commonalties. Three factors could be extracted that after a Kaiser normalization were rotated with the Varimax method. The first factor explains the fault detection processes with their related project and process metrics. It includes the measured process adherence within a project and year of handover, because these factors obviously relate to enforcing the inspection process. Metrics contributing to this factor are considered within this study. The second factor covers efficiency (i.e. "productivity" in terms of KStmt per PY) and overall quality metrics, namely normalized cost of non-quality, customer detected faults and faults per size. The third factor explains project parameters, such as size, efficiency, delivery accuracy and effort spent in the project. Obviously we have now to look for links among metrics clustered by the second and third factor.

5.2 Testing the Hypotheses

Since the three hypotheses are obviously related in their outcome, namely to reduce cost of non-quality and improve field performance, we will try to first summarize the achievements in one picture. Table 1 provides average values of the different process metrics in the discussed three-year timeframe. They look promising but can not immediately show detailed relations.

An overview of the investigated variables is given in Fig. 1 which shows the relations or root causes between the different investigated metrics. The three independent variables used in the three hypotheses are underlined. They are on the left side of the picture with arrows pointing towards the variables they influence. Spearman rank correlation coefficients are provided with each link which helps in finding the root causes easier than in a table with all correlation coefficients. The thick arrows pointing upwards or downwards indicate what would happen if one of the independent variables would change in the given direction. For instance, higher share of inspections would decrease normalized cost of non-quality. The three independent variables are indeed independent as they are not significantly correlated ($r < 0.4$). They are somehow re-

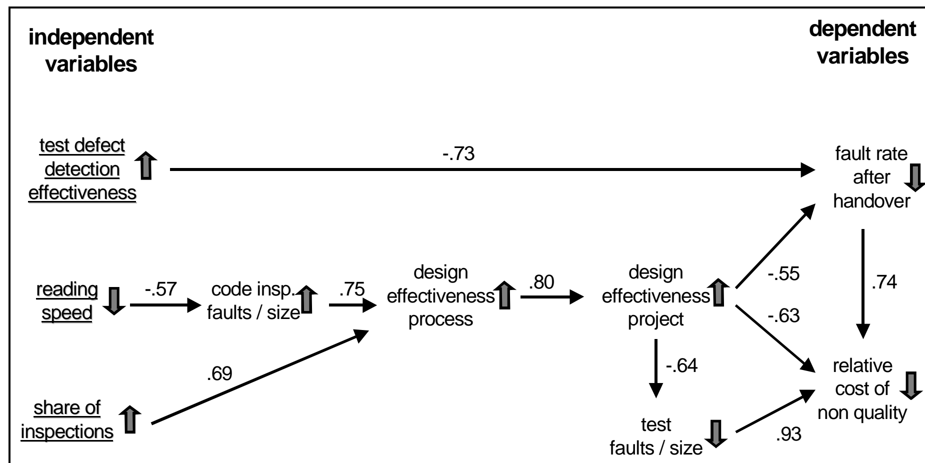
lated because the process adherence and the management attention that increased over time influence them all. However, no common underlying factor could be found.

Table 1: Achieving quality improvement targets. Average values are given for the three-year timeframe

	baseline	year 1	year 2	year 3
Reading Speed in code inspections [Stmt/Ph]	183	57	44	30
Faults found in Code Reading or Code Inspections [F/KStmt]	2	8	12	13
Faults found in Module Test [F/KStmt]	3	5	9	11
Design Effectiveness	17%	31%	46%	63%
Test Fault Coverage	57%	72%	83%	92%
Fault Detection in the Field *	100%	57%	28%	18%
Relative Cost per Fault *	100%	63%	55%	44%

* Start of Improvement Program taken as 100%

Figure 1: Relationships between the different measured variables. Values give the Spearman rank correlation coefficients. Underlined variables are independent variables that are controlled by the SPI program to achieve improvement for the dependent variables.

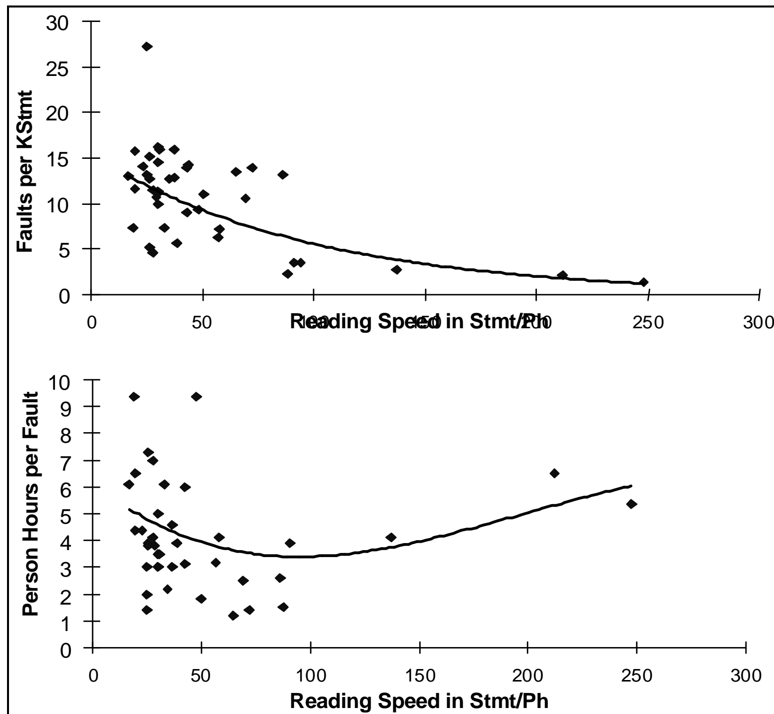


Hypothesis 1: There is an optimum code reading speed that balances the conflicting requirements for high fault detection rates (measured in design fault detection effectiveness) and low fault detection costs (measured in overall normalized cost of non-quality and amount of customer detected faults).

As a first overview fig. 2 summarizes experiences with reading speed that we collected during introduction of code inspections. Each dot refers to one project in the 3 year timeframe. Although faults per KStmt increase with reduced reading speed (Spearman r=0.57), efficiency is decreasing. The optimum speed based on the inserted trend lines which reflect best fit with a given residuum measure is around 90 Stmt/Ph. We ask in our process guidelines for a reading speed of 50...100 Stmt/Ph. The lower

speed applies for selected critical areas (ca. 10...20% of all changes), to handle the trade-off of cost and effectiveness.

Figure 2: Optimizing a process with process metrics. The lines are best-fit curves to indicate the trend. Although faults per KStmt increase with lower reading speed, efficiency is decreasing. The optimum is around 90 Stmt/Ph.



The rather big variance of efficiency values for lower reading speed in fig. 2 is caused by the total amount of faults within the code and their distribution. Since we summarize across projects, the amount of change and its distribution across modules impacts fault distribution and thus efficiency. For instance if changes are small, more context needs to be checked to see impacts on interfaces, while the detected faults mostly reside in the few changed statements. The result is lower efficiency.

Code reading speed directly impacts the amount of faults found per inspected size (Spearman coefficient r = - 0.57). The lower the code reading speed (i.e. within the given range that was investigated), the higher the amount of faults detected in the inspected code. This in turn impacts the fault detection effectiveness during design (Spearman coefficient r = 0.75). The more faults are detected with inspections the higher the overall effectiveness of early fault detection before start of test. Design fault detection effectiveness is split in effectiveness related to new or changed code and an overall percentage. They are labeled "process" (i.e. because this is clearly impacted by a changed process) and "project" (i.e. also including the reused code which occasionally would contribute to fault detection). Both these variables are highly cor-

related (Spearman coefficient r = 0.80).

A high removal rate of faults before start of test reduces the amount of faults found in test per new or changed size (Spearman coefficient r = -.64). The underlying assumption for this assertion is that the total amount of faults generated during coding is at least not increasing. We won't investigate at this point the impact of fault prevention as this activity needs more time to yield measurable large-scale benefits. Relative cost of non-quality is drastically influenced by a decreasing amount of faults per new or changed size, because these faults are the most expensive to detect and to repair (Spearman coefficient r = .93).

High effectiveness of fault detection before start of test is correlated with customer detected faults because inspections help detecting other classes of faults than what is found in test (e.g. wrong boundary conditions or overload could be found easier with inspections) (Spearman coefficient r = -.55). Increasing design fault detection effectiveness would also reduce the relative cost of non-quality (Spearman coefficient r = -0.63).

Another approach beyond mere correlation analysis is the analysis of variance. Since we cannot assume in all cases that the data is from a normal distribution we apply the Kruskal-Wallis test to tell if the mean rank of the projects with low reading speed is lower than that of the projects with higher reading speed. Fault-rate after handover yields H = 8.6 with p = .072. This means that the probability of assuming a random relationship between code reading speed and fault rate after handover is 7.2%. For relative cost of non-quality the Kruskal-Wallis test yields H = 8.6 with p = .072. The hypothesis can thus be accepted.

Hypothesis 2: Increasing the share of code inspections would improve design effectiveness and thus reduce normalized cost of non-quality and customer detected faults.

Performing more inspections before delivering the code to test helps in reducing the overall faults. This is reflected in the increase of design fault detection effectiveness as soon as more inspections are performed (Spearman coefficient r = .69). Share of inspections is the ratio of modules that were actually inspected following all rules related to a proper inspection process. From an increased share of faults detected during design onwards the root cause chain is the same as in hypothesis 1.

Hypothesis 3: Increasing test fault detection effectiveness (i.e. percentage of remaining faults being detected during integration) would reduce normalized cost of non-quality and customer detected faults. Increasing test fault detection effectiveness directly reduces the amount of faults found after handover (Spearman coefficient r = -.73). This is obvious as long as test is actually detecting those fault types that later in the field would cause performance problems and failures. It is not necessarily the case because still the art of testing is to try those test cases that would show failures that later harm. Clearly there are classes of faults that although never been detected still would not cause problems - and vice versa.

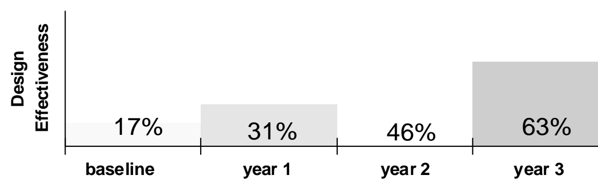
The cost per fault is steeply increasing towards handover. In the field increases towards a factor 10..20 compared to fault detection with code inspections are typical. This means that the more faults are found upfront the lower the cost of non-quality in total. This can also be seen by the strong impact of fault rate after handover on cost of non-quality (Spearman coefficient r = .74).

Design fault detection effectiveness is one of the key variables used in this study. It measures the share of faults found before start of test versus the total amount of faults. Increasing design fault detection effectiveness over a three year timeframe is indicated in Fig. 3. At the start of the improvement program 17% of all faults were detected before the start of test, while after 3 years almost two thirds of all faults are detected upfront.

6. ROI Calculation

ROI (return on investment) is a critical and misleading expression when it comes to development cost or justification of new techniques [1,2]. Too often heterogeneous cost elements with different meaning and unclear accounting relationships are combined to one figure that is then optimized. For instance reducing "cost of quality" that include appraisal cost and prevention cost is misleading when compared with cost of nonconformance because certain appraisal cost (e.g. module test) are a component of regular development. Cost of nonconformance on the other hand is incomplete if only considering internal cost for fault detection, correction and redelivery because they must include opportunity cost due to rework at the customer site, late deliveries or simply binding resources that otherwise might have been used for a new project.

Figure 3: Design fault detection effectiveness improvement over three years



ROI is difficult to calculate in software development. This is not so much any more due to not having collected effort figures but rather by distinguishing the actual effort figures that relate to investment (that would have otherwise not been done) and the returns (as difference to what would have happened if not having invested). For many years ROI data was reported but in most cases not backed up by real data, or where real data existed, it was counter to the current mainstream viewpoints [8]. Only recently first studies have been published that try to compare results of software process improvement activities [1].

We will try to provide insight in a practical ROI calculation (Table 2). Given an average sized development project and only focusing on the new and changed software without considering any effects of fault preventive activities over time, the following calculation can be derived. The effort spent for code reading and inspection activities increases by 1470 Ph. Assuming a constant average combined appraisal cost and cost of nonperformance (i.e. detection and correction effort) after coding of 15

Ph/Fault, the total effect is 9030 Ph less spent in year 2. This results in a ROI value of 6.1 (i.e. each additional hour spent during code reading and inspections yields 6.1 saved hours of appraisal and nonperformance activities afterwards).

Table 2: ROI calculation of process improvements with focus on code inspections *(fault preventive activities are not considered for that example)*

	baseline	year 1	year 2
Reading Speed [Stmt/Ph]	183	57	44
Effort per KStmt	15	24	36
Effort per Fault	7.5	3	3
Faults per KStmt	2	8	12
Effectiveness [% of all]	2	18	29

Average Project: 70 KStmt resulting in 3150 Faults			
Effort for Code Reading / Insp. [Ph]	1050		2520
Faults found in Code Reading / Insp.	140		840
Remaining faults after Code Reading	3010		2310
Correction effort after Code Reading / Insp. [Ph] (based on 15 Ph/F average correction effort)	45150		34650
Total correction effort [Ph]	46200		37170

ROI = saved total effort / add. detection effort	6.1

We had the following experiences with ROI calculations:

- It is better to collect the different effort figures during a project than afterwards
- Activities related to distinct effort figures must be defined (activity based costing helps a lot)
- Cost and effort must not be estimated, but rather collected in projects (typically the inputs to the estimation are questioned until the entire calculation is not acceptable any more)
- Detailed quality cost are helpful for root cause analyses and related fault prevention activities
- Tangible cost savings are the single best support for a running improvement program
- Cost of nonperformance is a perfect trigger for a SPI program
- There are many ”hidden” ROI potentials that are often difficult to quantify (e.g. customer satisfaction; improved market share because of better quality, delivery accuracy and lower per feature costs; opportunity costs; reduced maintenance costs in follow-on projects; improved reusability; employee satisfaction; resources are available for new projects instead of wasting them for firefighting)
- There are also hidden investments that must be accounted (e.g. training, infrastructure, coaching, additional metrics, additional management activities, process maintenance)

Not all ROI calculations are based on monetary benefits. Depending on the business goals it can as well be directly presented in improved delivery accuracy, reduced lead time or higher efficiency and productivity.

7. Conclusions

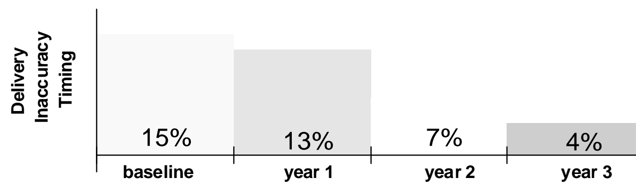
We have presented some extracts from the SPI initiative of Alcatel's Switching Systems Division. While this SPI program is ongoing and will surely yield additional improvements not mentioned here, it should be clear that SPI - especially because of its rather long elapse time until sustainable results are achieved - must be entirely based on a comprehensive and integrated improvement approach. Three hypotheses have been validated in this study, namely

- reducing reading speed during code inspections improves design effectiveness and thus reduces normalized cost of non-quality and customer detected faults;
- increasing the share of code inspections after coding and before test improves design effectiveness and thus reduces normalized cost of non-quality and customer detected faults;
- increasing test fault detection effectiveness (i.e. percentage of remaining faults being detected during integration) reduces normalized cost of non-quality and customer detected faults.

To further reduce cost of non-quality it was obvious that SPI also had to focus on product line impacts and not only on a per project approach. This was yet another lesson learned that typically is not mentioned in current empirical SPI studies. We introduced a synchronization approach to simultaneously trigger corrections in all impacted customization projects. Further we started an alignment of product variants while still preserving the desired operator-specific features asked by different customers operating in the same markets.

Although we focused in this study on achievements related to product quality, other results became visible in parallel. With improved product quality and reduced rework, we were able to focus resources according to customer needs and therefore improve delivery accuracy of the various customer projects (Fig. 4).

Figure 4: Delivery inaccuracy (in terms of achieved handover date vs. plan) improvement over three years



Software process improvement is now a big issue on the agenda of all organizations with software as a core business. As such it is also a major research topic, that may continue to grow in importance well into the 21st century. However, some software technologies have a much shorter lifetime and for sure the management attention is focused rather on short-term achievements with impact to the scorecard. Unless tangible results can be achieved in the related short timeframe, interest in SPI will quickly

wane.

References

[1] McGarry, F. et al: Measuring Impacts Individual Process Maturity Attributes Have on Software Products. Proc. 5. Int. Software Metrics Symposium. IEEE Comp. Soc. Press, pp. 52-60, 1998..

[2] Bassin et al: Evaluating Software Development Objectively. *IEEE Software*, Vol. 15, pp. 66-74, 1998..

[3] Wohlwend, H. and S.Rosenbaum: Schlumberger's Software Improvement Program. *IEEE Trans. Software Engineering*. Vol. 20, No. 11, pp. 833-839, Nov.1994.

[4] Wigle, G.B.: Practices of a Successful SEPG. *European SEPG Conference 1997*. Amsterdam, 1997. More in-depth coverage of most of the Boeing results in: G.G.Schulmeyer and J.I.McManus, Ed.: Handbook of Software Quality Assurance, 3. ed., Int. Thomson Computer Press, 1997.

[5] Grady, R.B.: *Successful Software Process Improvement*. Prentice Hall, Upper Saddle River, 1997.

[6] Diaz, M. and J.Sligo: How Software Process Improvement Helped Motorola. *IEEE Software*, Vol.14, No. 4, pp. 75-81, Sep. 1997.

[7] Dekleva, S. and D.Drehmer: Measuring Software Engineering Evolution: A Rasch Calibration. *Information Systems Research*. Vol. 8, No. 1, pp. 95-105, Mrc. 1997.

[8] Fenton, N. E. and S.L. Pfleeger: *Software Metrics: A Practical and Rigorous Approach*. Chapman & Hall, London, 1997.

[9] Perpich, J.M., et al.: Anywhere, Anytime Code Inspections: Using the Web to remove Inspection Bottlenecks in Large-Scale Software Development. *Proc. Int. Conf. on Software Engineering*, IEEE Comp. Soc. Press, pp. 14-21, 1997.

[10] Perry, D.E., H.P.Siy and L.G.Votta: Parallel Changes in large Scale Software Development: An Observational Case Study. *Proc. Int. Conf. on Software Engineering*, IEEE Comp. Soc. Press, pp. 251-260, 1998.

[11] Ebert, C.: Technical Controlling and Software Process Improvement. *Journal of Systems and Software*. Vol. 46, pp. 25-39, 1999.

[12] Ebert, C., J.Altenhoener, J.DeMan: Integrating Process Improvement and Product Engineering. Proc. 5. Annual European Software Engioneering Process Group Conference. Amsterdam, 5.-8. Jun. 2000. ESPI Foundation, www.espi.co.uk. Jun. 2000.