An Empirical Study of Software Change: Origin, Acceptance Rate, and Functionality vs. Quality Attributes

Parastoo Mohagheghi[1,2,3], Reidar Conradi[2,3]
[1]*Ericsson Norway-Grimstad, Postuttak, NO-4898 Grimstad, Norway*
[2]*Department of Computer and Information Science, NTNU, NO-7491 Trondheim, Norway*
[3]*Simula Research Laboratory, P.O.Box 134, NO-1325 Lysaker, Norway*
parastoo@idi.ntnu.no, conradi@idi.ntnu.no

Abstract

The paper presents results from an empirical study of change requests in four releases of a large-scale telecom system that is developed incrementally. The results show that earlier releases of the system are no longer evolved. Perfective changes to functionality and quality attributes are most common. Functionality is enhanced and improved in each release, while quality attributes are mostly improved, and have fewer changes in forms of new requirements. The share of adaptive/preventive changes is lower, but still not as low as reported in some previous studies. Data for corrective changes (defect fixing) have been reported by us in other studies. The project organization initiates most change requests, rather than customers or changing environments. The releases show an increasing tendency to accept change requests, which normally impact project plans. Changes related to functionality and quality attributes seem to have similar acceptance rates. We did not identify any significant difference between the change-proneness of reused and non-reused components.

1. Introduction

An important study object in empirical software engineering is *software maintenance*, being prevalent and thus costly in most software systems. Earlier studies have tried to study maintenance aspects, such as the ratio between different categories of maintenance activities, the origin of changes, or the impact of changes. These questions need updated answers given the emergence of new development approaches, such as incremental and iterative development. While incremental means that the project scope is (discovered and) covered in steps, iterative means that the developed assets are improved gradually during iterations. As many software projects are developed incrementally and iteratively, the subject of software change is relevant not only in the maintenance phase, but also in evolution between releases. Another aspect is the increasing use of component-based development (CBD) and software reuse, and the question that whether maintainability have improved.

This article describes the results of analyzing change requests (CRs) from four releases of a large telecom system developed by Ericsson over a three-years period. CRs cover any change in the requirements or assets from the time of requirement baseline. We study some related factors, and assess five hypotheses, concerning the category of changes (perfective etc.), their origin, their acceptance rate, and their relation to reuse. We look at perfective, adaptive and preventive changes that characterize evolution. Corrective changes have been analyzed by us elsewhere [12].

The results show that earlier releases of the system are no longer evolved, and functionality is enhanced and improved in each release. Quality attributes are mostly improved, and have fewer changes in forms of new requirements. Most CRs are initiated internally by the project organization, and the acceptance rate of CRs has been increasing over time. When it comes to reuse, there was no significant difference between the change-proneness (number of CRs per KLOC) of reused and non-reused components. However, our earlier study of corrective changes shows that reused lower-level components are more stable (less modified code), and have fewer defects than are non-reused ones.

The remainder of this paper is organized as follows. Section 2 includes a description of some related work.

Section 3 presents the Ericsson context, and the available data. Section 4 describes the research method, and hypotheses. Section 5 presents the results, which are discussed further in Section 6. Section 7 contains the conclusion.

2. Related work

2.1. Concepts for software change

Lehman's first law of software evolution says that "an E-type program that is used must be continuously adapted else it becomes progressively less satisfactory" [7]. An E-type program is a software system that solves a problem in the real world. The growth of a system may be observed (measured) in many ways, for example by the amount of modified code between releases or per interval of time, the number of modules, the volume of change-logs, or even the number of system releases per time unit [14]. In other words, the granularity of change data varies. While lower level granularity provides most detailed information, it is more difficult to gather.

When a software system is still under development, requirements of the system may change, and requirement volatility may impact the project performance (e.g. schedule or cost overruns) or the quality of the software product (e.g. increasing defect density). These impacts have been subject of empirical studies; see e.g. [5] [9] [18]. Other studies investigate modifications to software after it has gone into production; i.e. the maintenance phase, e.g. [10] [14]. The main challenge of empirical studies in this field is to have access to consistent records of software changes over time due to the longitudinal nature of the study.

Changes may be categorized as corrective, adaptive, perfective, or even preventive. *Corrective maintenance* refers to defect repair. *Adaptive maintenance* means adapting to a new environment or a new platform. *Perfective maintenance* is both used for implementing new or changed requirements, and for improving system performance; i.e. both functional enhancements and non-functional optimizations [16]. *Preventive maintenance* is sometimes used about internal restructuring or reengineering in order to ease later maintenance. Sometimes these terms are defined differently, making comparison of studies difficult. For example, Mockus et al. used adaptive maintenance to cover enhancements, and perfective to cover optimization [10]. Some others argue that it is maintenance when we correct errors, but it is *evolution* when we respond to other changes. Bennett and Rajlich distinguish between development, evolution and maintenance [4]. In their terminology, development lasts until a system is delivered to production. When a system is in production but still growing, it is in the evolution phase. They also provide a model for incremental evolution called for the *versioned staged model* shown in Figure 1, comparing to the simple staged model for evolution. In this model, after release of a version it is no longer evolved, only serviced. All new requirements will be placed on the new version.

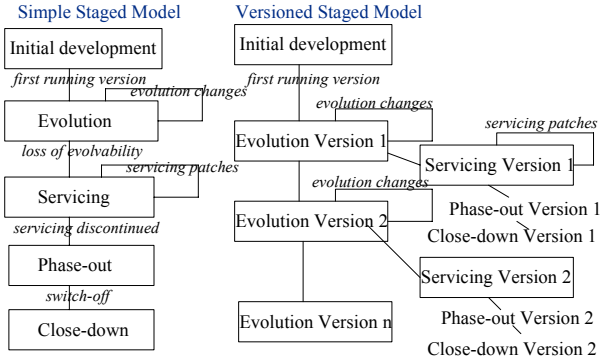

Figure 1. The incremental (versioned staged model) of software evolution from [4]

2.2. Results of previous studies

Damian et al. [5] describe results of a survey on the impact of improving the *pre-delivery* requirement engineering process on several factors. Zowghi and Nurmuliani [18] have similarly performed a survey among 430 software-developing companies in Australia on the impact of changing requirements on project performance regarding schedule and cost. The survey results show a negative correlation between the degree of requirement volatility, and both schedule and cost performance. Earlier work by Stark et al. [17] confirms this result.

One of the first studies on the distribution of *post-delivery* maintenance activities is reported in 1978 [8]. Based on the results of a survey among maintenance managers, Lientz et al. reported that 17.4% of the maintenance effort was categorized as corrective, 18.2% as adaptive, 60.3% as perfective, and 4.1% as other. Jørgensen [6] has observed that if the amount of corrective work is calculated based on interviews, it will be as twice as the actual work reported in such logs. I.e. the amount of corrective work may be exaggerated in interviews.

Schach et al. [14] have analysed detailed data from 3 software products on the level of modules, and

change-logs. The products were a 12 KLOC real-time product, a subset of Linux consisting of 17 kernel modules and 6506 versions, and GCC (GNU Compiler Collection) consisting of nearly 850 KLOC. For these products, the distribution of maintenance categories was over 50% for corrective, 36-39% for perfective, and 2-4% for adaptive maintenance. In other words, the distribution is very different from the results reported by Lientz et al.

Mockus et al. [10] have used historical data of change requests of a multi-million-line telecom software system, and report the results both on the change-log level and on LOC (Lines of Code) added, deleted or modified. They report that adding new features (perfective changes) accounted for 45% of all changes, followed by corrective changes that accounted for 34%, while restructuring of the code accounted for 4% of changes (mostly preventive changes). Although comparisons of results are not easy between these two studies because of different categorizations, both indicate a large portion of corrective changes, as well as perfective changes for new requirements.

Algestam et al. [1] report a study in Ericsson of a large telecom system. Reusing components and a framework resulted in increased maintainability evaluated in cost of implementing change scenarios, improved testability, easier upgrades, and also increased performance. The impact of software reuse, especially exploiting COTS (Commercial-Off-The-Shelf) components, is studied e.g. in [2], and [3].

Organizations typically have a change management process to accommodate for requirement or artifact changes. In incremental development, each release may have changes in requirements or deliveries, and is undergoing an evolution phase. Evolution of a system should therefore be studied in two phases: during a release, and between successive releases. None of the studies above have taken the step to the versioned staged model shown in Figure 1, or separated these two phases, that may have different characteristics. The studies have also not separated functionality and quality attributes.

2.3. Research questions

Costs related to software evolution and maintenance activities can exceed development cost. Changes influence project performance and product quality. The impact of development approaches on software evolution and maintenance is also important to assess. Incremental and CBD are new approaches with few empirical studies on their impact on software evolution and maintenance. The following research questions are identified for this study:

RQ1: Do the majority of changes originate from external factors or from the project organization itself?

RQ2: Are changes mostly due to functional enhancements, or optimization of quality attributes?

RQ3: What is the impact of changes in terms of effort, size of modified code, or type of components?

RQ4: In which phase of the project are changes mainly introduced?

3. The Ericsson context

Ericsson in Grimstad-Norway has developed software for several releases of two large-scale telecom systems. The systems are characterized by high performance, high availability, scalability, frequent hardware and software upgrades, and distribution of software over multiple processors.

3.1. Overview of the products and the development process

The first system was originally developed to provide packet data capability to the GSM (Global System for Mobile communication) cellular network. A later recognition of common requirements with the forthcoming WCDMA system (Wide-band Code Division Multiple Access) lead to reverse engineering of the original software architecture to identify reusable parts across the two systems. The two systems (or products) called for *A* and *B* in Figure 2, are developed incrementally, and new features are added to each release of the systems. The architecture is component-based, and all components in our study are built in-house. The higher-level components are subsystems (consisting of blocks, which are the lower level components) with almost 90 KLOC on the average. Both systems *A* and *B* contain top-level components respectively from Application *A* or *B*, as well as shared and reused components from the business-specific and middleware layers. At the bottom, there is a Wireless Packet Platform (WPP) serving as a pre-provided operating system.

The development process has evolved as well: The initial development process was a simple, internally developed one, describing the main phases of the lifecycle and the related roles and artifacts. After the first release, the organization decided to adapt the Rational Unified Process (RUP) [13]. Each release goes through 5-7 iterations. Multiple programming languages are used; Erlang and C are dominant, Java is used for GUIs, and Perl and other languages are used

for minor parts. The size of each system (not including the system platform) is over 1000 NKLOC (Non-Commented Kilo Lines Of Code measured in equivalent C, see [12] for more details) in the last releases. Several hundred developers in different Ericsson organizations have been involved in developing, and testing the releases. Our data covers 4 releases of system *A*, where business-specific and middleware components are reused in two applications in release 4.

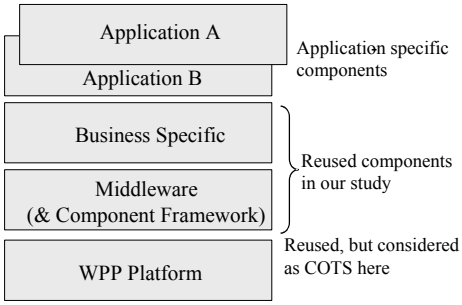

Figure 2. High-level architecture of systems A and B

3.2. Change Requests (CRs)

RUP is an incremental software process with four phases: Inception, Elaboration, Construction, and Transition. Each phase concludes activities from several workflows; i.e. Requirement Management (RM), Analysis and Design (A&D), Implementation and Test, and may include one or several iterations. The original product, and project requirements for each release are stated in a textual document called the ARS (Application Requirement Specification). Requirements in the ARS are refined iteratively during the inception and elaboration phases, resulting in artifacts such as use case models, use case specifications, supplementary specifications documents (for non-functional requirements), and statement of compliances towards standards that the system must fulfill. The ARS and the detailed set of the requirements are baselined at the end of initial iterations, and again at the end of the elaboration phase. Changes after this milestone are proposed through formalized CRs. Examples of proposed changes are:

- Add, change or delete functionality (perfective functional enhancements).
- Propose an improvement of a quality attribute (perfective quality optimizations).
- Implement a cost reduction.
- Solve an anticipated problem with major design impact (preventive).

CRs reflect coarse-grained changes that should be formally approved, and can significantly impact the contents of a release, cost or schedule. Small changes in implementation or documentation are done in each release by respective teams without issuing a CR. Figure 3 shows the phases and states in the CR handling process. CCB stands for the *Change Control Board*, who is responsible for taking decisions for approval or rejection of a CR. While the organization keeps a statistics over the number and states of the CRs, no systematic study of the CRs is done previous to this study.

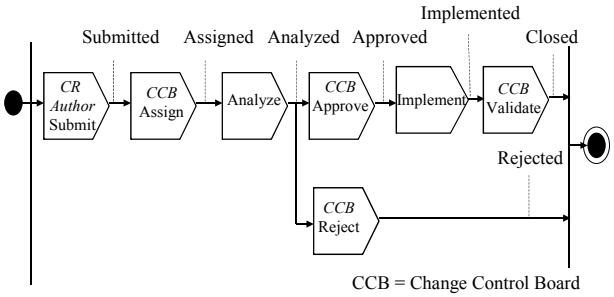

Figure 3. Flow of a CR and its states

CRs were at the beginning written in FrameMaker. Recently, these are written in MS-Word. The templates have changed several times, and not all fields are filled-in. The current template includes the following information: Title, revision history, baseline affected, documents/artifacts affected, description of the current situation and the proposed change, consequences of acceptance or rejection, and an estimate of the needed effort to implement the CR.

Note that CRs may be issued pre- or post-delivery. Our CRs fall in the category of perfective (enhancements and optimizations), adaptive (towards the WPP platform or standards), and preventive changes (reorganizations).

3.3. Change Request data

The first set of CRs was extracted from the version control system on January 2003 by a team of two NTNU students. This set included 165 CRs, issued from June 2000 to June 2002. 15 of these CRs handled in fact deviations to the process, and were omitted from the rest of this study. The status and a short summary of the CRs are given in html pages controlled

by the CCB. We created a tool in C# that parsed the html pages, and inserted relevant fields in a Microsoft SQL database. For other fields that were not given in the html pages, the students read the CRs and inserted the data manually in the database. The first author checked these data later, and one or two fields in totally 24 CRs (of 150 such) were changed after this second check, being considered a small modification. A second set of CRs was extracted in November 2003 by the first author, and included 19 CRs issued from October 2002 to November 2003. These CRs were inserted manually in the database. Thus, we totally have 169 CRs for 4 releases of system *A*, as shown in Table 1. Release 3 did not include any new functionality, but was a new configuration in order to separate nodes in the system.

Table 1. Overview of all 169 CRs

	Rel. 1	Rel. 2	Rel.3	Rel. 4
Pre-delivery	10	37	4	99
Post-delivery	0	0	0	19
SUM	10	37	4	118

The number of CRs has increased dramatically as the product evolves from release 1 to 4. This increase is partly because the CR handling process has matured over time. For instance, some changes of release 1 were handled informally. However, because of the growing complexity of the releases, it is not unexpected that there would be more changes to the requirements or products over time, and the time frame in which a product is "under evolution" increases. We notice that evolution of releases 1-3 has stopped. These releases were delivered to the market, put in the servicing phase, and will be phased out after a while. Release 4 still evolved at the time of study as new CRs were issued. We also studied the date of initiation of CRs, which showed that most CRs in each release are initiated in a short time after requirement baselining.

Data on estimated cost or needed effort is not used in the study since we don't have data on actual cost or effort. Data on effected components is coarse-grained as discussed later. Otherwise, the data set is considered to be reliable for the study.

We have described results of a study on corrective maintenance (Trouble Reports or TRs) in [12]. The data for that study covered TRs for these 4 releases until January 2003. We mention that the number of TRs were 6 for release 1, 602 for release 2, 61 for release 3, and 1953 for release 4. Again, not all TRs for release 1 were stored in this database. We note the same increase in the number of TRs as for CRs as shown in Table 1.

4. The research method and hypotheses

We tested five hypotheses on the available data. Choosing hypotheses has been both a top-down, and a bottom-up process. Some goal-oriented hypotheses were chosen from the literature (top-down), to the extent that we had relevant data. In other cases, we pre-analyzed the available data to find tentative relations between data and possible research questions (bottom-up) as in an exploratory research. Table 2 shows the hypotheses, their relations to research questions (RQ) defined in Section 2.3, and their grouping.

Table 2. The five research hypotheses

Hyp. group	Hyp. Id	Hyp. Text	RQ
Origin	**H01**	Pre-implementation, and post-implementation CRs have equal proportions.	4
	HA1	Most CRs are for post-implementation changes.	
	H02	Quality attributes, and functionality have equal proportions of CRs.	2
	HA2	Most CRs are due to quality attributes, rather than to functionality.	
	H03	Customers and changing environments initiate as many CRs as the project organization.	1
	HA3	Customers and changing environments initiate most changes.	
Acceptance	**H04**	CRs that are accepted, and CRs that are rejected have equal proportions.	3
	HA4	Most CRs are accepted.	
Reuse-CBD	**H05**	Reused and application components are equally change-prone.	3
	HA5	Application components are more change-prone than are reused ones.	

A short description of background for each hypothesis is given below. Apart from assessing the five hypotheses, we will study some relationships

between these, e.g. what class of CRs are mostly accepted or rejected.

H01-HA1: CRs have a field that indicates whether a CR specifies a change in requirements (new, modified, or removed requirement as stated in the ARS or other requirement specification documents) *before* implementation, or a change to the product or documentation *after* a requirement is first implemented and verified. Some CRs have left this information out, and are instead classified by us. **H01** states that the proportions of CRs for requirement changes, and CRs for modifications of the product are equal. The alternative hypothesis **HA1** states that most changes are post-implementation changes to the product. As the organization use much effort in the CR handling process, it is important to assess whether this effort is because of unstable requirements, or iterative improvement of solutions.

H02-HA2: CRs may also be categorized on whether they deal with functionality or with quality (non-functional) attributes. This information is extracted from the description and the consequences of approval or rejection. Earlier studies do not differ properly between these two sub-categories of perfective changes. The practice is that functional requirements are specified well, and thus changes in those would be more obvious. **H02** states equal proportion, while the alternative hypothesis **HA2** states that most changes are related to quality attributes, rather than to functionality, in line with HA1.

H03-HA3: CRs may be initiated internally by the project organization in order to improve or enhance the product, or externally by the customers or due to changing environments (external factors). Damian et al. [5] write that changes in requirements often arise from external events originating outside the organization, such as unpredictable market conditions or customer demands. **H03** states that there is no difference between proportions of CRs in these two groups. **HA3** states that the Domain's claim is true.

H04-HA4: Waterfall development requires stable requirements, while incremental approaches are more open to changes. We want to assess the stability of requirements, and the product. As we don't have data on the actual impact of CRs in terms of modified Lines of Code in some normalized form, it is difficult to assess the absolute impact. Therefore we chose to study the share of CRs being accepted or rejected. **H04** states that the proportions are equal, while **HA4** states that most CRs are accepted. If most CRs get accepted and implemented, the organization should be prepared to account for additional resources to handle and implement these CRs.

H05-HA5: We define change-proneness as #CRs/KLOC. Components in the business-specific and middleware layers were reused in two systems in release 4. **H05** states that there is no difference in change-proneness of these components. The alternative hypothesis would be that one component type is more change-prone than is the other. As application components are "customer-close", we may assume that these are more change-prone, as stated in **HA5**.

5. Data analysis and assessment of hypotheses

Table 3 shows a summary of the assessment of hypotheses. Here *NR* stands for *Not Rejected*, while *R* means *Rejected*. The remainder of this section describes the detailed results.

Table 3. Assessments of hypotheses

Hyp. Id	Result	Conclusion
H01	**NR**	The difference between proportions of CRs issued before or after implementation is not significant.
H02	**R**	Most perfective changes are due to quality attributes, not functionality.
H03	**NR**	Most CRs are originated inside the organization.
H04	**R**	Most CRs are accepted and implemented.
H05	**NR**	Reused and application components are equally change-prone.

We used Microsoft Excel and Minitab in statistical tests. The confidence level is 95% in all tests, which means that we reject the null hypotheses if the observed significance level (P-value) is less than 5%. The 5% significance level is the default value in most tools, and in practice much higher P-values may be accepted. We therefore present the distributions, and the P-values to let the reader decide, as well as presenting our conclusions.

5.1. H01-H03: Origin of CRs

Table 4 below shows a classification of CRs for all four releases. We see that most changes are optimizations of solutions, followed by new requirements after baseline. We could also compare the share of requirement changes (47.3%) before implementation, to the share of later modifications in solutions or documentation (52.1%). We tested whether the proportions are equal vs. greater

proportion of CRs for modifications (one proportion test in Minitab for proportion=0.5 vs. proportion>0.5). The P-value is 0.322; i.e., it is 32% possible that the observed difference is by chance. The conclusion is that we cannot reject **H01**.

Table 4. Distribution of 169 CRs over pre- and post-implementation changes

	New Req.	Modified Req.	Removed Req.	Modified Solution	Modified Doc.	Other
No	46	25	9	70	18	1
%	27.2	14.8	5.3	41.4	10.7	0.6

Table 5 shows the distribution of CRs over evolution categories, and a more detailed distribution over CR-focus or *reason*. Note in Table 5, that the sum of numbers is 187 (and the sum of percentages is over 100%), as 18 of 169 CRs have indicated *two* reasons for requesting the change. Also note that preventive changes to improve file structure or to reduce dependencies between software modules and components may later impact quality attributes such as maintainability. For the systems in study, there is great emphasis on quality attributes, reflected in the large number of CRs for this group.

For perfective CRs, the proportion of functional and quality attributes CRs are 35% (40/114) and 65% (74/114). We performed a one proportion test in Minitab that gives a P-value of 0.01, which means that we can be 99% sure that the difference is significant, and may reject **H02** in favor of **HA2**.

The contents of Tables 4, and 5 are combined in Figure 4. Perfective functional CRs have almost equal distribution between new requirements, and modified solutions. Perfective quality attributes and preventive CRs are mostly modified solutions. Adaptive CRs are mostly new or modified requirements

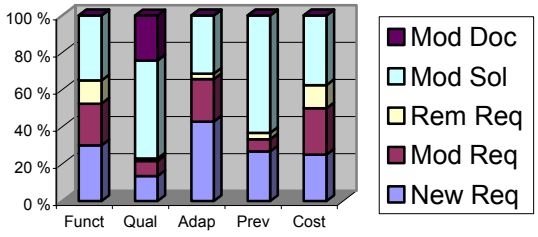

Figure 4. Evolution categories vs. requirements or solutions

Table 5. Distribution of CRs over maintenance categories and CR-focus

Evolution category	CR-focus (*reason*)	No.	Accepted	Examples
Perfective/ Functional	Functionality	40	26	Business or middleware functions
Perfective/ Quality Attributes SUM = 74	Performance	29	16	Storage, throughput
	Documentation	21	13	Understandability, customer documents
	Availability	11	7	Increasing up-time
	Testability/ Maintain-ability	11	6	Remote testing, monitoring alarms
	Security	2	1	Protecting contents
Adaptive SUM = 35	Standards	18	13	Compliance, new standards
	Interfaces	2	1	External interfaces
	WPP-upgrades	13	7	Change the platform
	WPP-adaptation	2	2	Adapting code to WPP changes
Preventive SUM = 30	System	19	7	Builds, configuration
	Re-structuring	11	7	Models, dependencies between entities, file structure
Other	Cost	8	3	Saving money/effort
TOTAL SUM		187		

23 of 169 of the CRs are issued because of customer demands. If we exclude these CRs, and the 35 CRs due to adaptive changes, the overwhelming group (111 of 169) is still CRs that originate inside the project organization to enhance or optimize the products. The one proportion test in Minitab shows that the proportion of CRs due to external factors (customers and changing environments) is 34%, and the P-value is 1.000. Hence the proportion of CRs due

to external factors is definitely lower than is the other group, opposed to **HA3**.

5.2. H04: Acceptance rates of CRs

Figure 5 shows the acceptance rates of the CRs as of November 2003. 99 CRs were *accepted* (approved, implemented, or closed), while 68 CRs are *rejected* (including those cancelled). Performing a one-proportion test gives a P-value of 0.015, which means that the difference is significant. Hence, **H04** is rejected in favor of **HA4**.

Figure 6 shows the distribution of CR classes and accepted vs. rejected states. All CRs that requested to remove a requirement are accepted, while the group that has the highest rejection rate is new requirements.

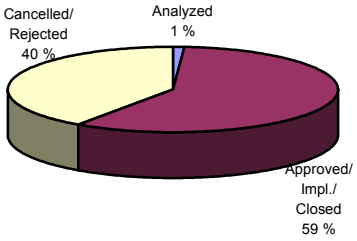

Figure 5. Acceptance rate of 169 CRs

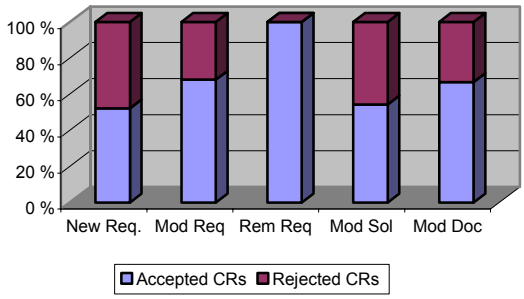

Figure 6. Acceptance rates and CR classes

We performed Chi-Square tests to study whether there is any relation between CR categories as defined in Table 5, and acceptance rates. The P-value of the test is 0.253 (DF=4), which indicated no relation. However, it is interesting to note that the maximum acceptance rate is for those with CR-focus Standards (72%), while the minimum rate is for those with System and Cost (38% both). The others vary between 50 and 60%. As our system should comply with international standards to be competitive and to inter-work with systems from other telecom operators, it is not surprising that changes due to Standards are mostly accepted. However, the low acceptance rate for Cost is surprising. The number of associated CRs is only 8, although many other CRs also impact cost (for example CRs that ask for removal of requirements). We cannot conclude otherwise that (low) cost is not a strong enough reason to accept a CR by itself.

Finally, we want to analyse whether the four releases vary significantly in acceptance rates of the CRs. Release 3 only has four CRs and hence cannot contribute to any significant conclusion. The overall results show that the acceptance rates of CRs have been increasing over the lifetime of the product as shown in Figure 7.

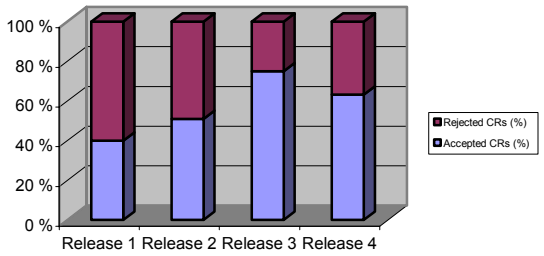

Figure 7. Relative distribution of CR states over the product releases (total of 169 CRs)

The organization has already studied *requirement volatility* in high-level requirements (those stated in the ARS), i.e. if they change after baseline. While this rate is 10% for Release 1, it is almost 30% for Release 3. I.e. both results indicate that the product is getting more change-prone over releases, or more changes are allowed.

5.3. H05: Software Reuse and CBD

Reuse of these components started from release 4, when development of system *B* started. Regrettably, only 81 of 118 CRs of release 4 have registered athe affected component name in the CR. Besides, CRs only register higher-level components, i.e. subsystems that consist of several related blocks.

System *A* consists of 3 application subsystems, 4 subsystems in the business-specific later, and 6 subsystems in the middleware layer (thus 10 reusable subsystems) in this release. Figure 8 shows the distribution of #CRs per KLOC for subsystems. We have one outlier, which is a small subsystem, handling configuration and tools, and which is left out from the statistical test. We performed a two-tailed t-test, which showed no significant difference in means for reused vs. non-reused components; i.e. P(T<=t) two-tail was 0.61, and **H05** can not be rejected.

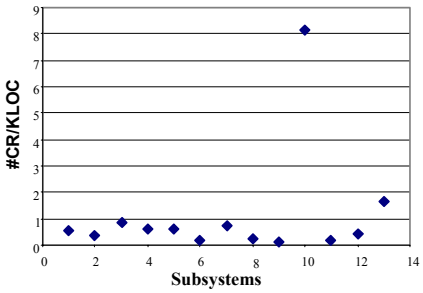

Figure 8. #CRs/KLOC for all 13 subsystems

6. Discussion

We comment each of the five hypotheses below.

H01: Most CRs are issued in order to optimize, and modify the product or documentation, rather than changing the requirements, but the difference is not significant. We need further analyses of the CRs to conclude whether the organization could save some effort by better quality assurance of the solutions.

H02: Although CRs issued to change, enhance or remove functionality account for the largest single group of CRs, quality attributes' related CRs are the largest as a group. This result highlights that that these attributes are optimized over time, and these improvements will have great impact of the evolution of the products.

H03: The results show that the project organization initiates most CRs for enhancing or optimizing the product.

H04: The results show that most CRs are accepted, especially those that request modification of a requirement or documentation, or removal of a requirement. New requirements need resources for implementation, and modification of previous solutions may be considered as too time-consuming compared to the perceived benefits, and therefore are more rejected. There was no significant difference in acceptance rates of the functional vs. quality attributes CRs. One interesting result is that the acceptance rate has been increasing in releases, and this may impact the precision of plans. The organization has already realized that planning precision has been decreasing. In [11] we described that incremental development opens for (more) changes in requirements, and this study demonstrates this.

H05: We could not observe any significant difference between reused, and non-reused components in number of CRs per KLOC. We have shown in [12] that reused components (as blocks) are more stable in terms of volume of code modified between releases, and more reliable in the terms of the number of Trouble Reports per KLOC. Together, these results quantify the benefits of reuse.

The study raises some interesting questions as well: Does the organization take the perhaps costly decision to baseline requirements too early, while the product still undergoes dramatic evolution? Could the number of changes be predicted for future releases using #CRs/KLOC from earlier releases?

Lastly, We have identified the following validity threats:

Construct validity: Most data categories are taken from the literature, and represent well-known study concepts. We used maintenance categories for all changes during development *after* requirement baseline. Previous works study changes post-delivery in the maintenance phase.

Internal validity: The biggest threat is that we ignore many CRs with no subsystems given in **H5**.

External validity: The study object is a large telecom system during three years of development. The results should be relevant and valid for similar systems and organizations, but not e.g. for web-based systems with very high change rates.

Conclusion validity: In **H5**, we have too little data caused by coarse-granular subsystems, . Otherwise, the data material is sufficient to draw valid conclusions.

7. Conclusion and future work

We defined 5 research questions in Section 2.3, and related them to the research hypotheses in Table 2. The results of the analyses are used to answer these:

RQ1 & RQ2 (origin): Most changes originate from the project organization in order to improve quality, and enhance functionality. The share of the first group is higher. The practice indicates iterative realization, and improvement of quality attributes, but functionality is also improved in a lower degree.

RQ3 (impact): CRs are not supplied with the actual cost of implementing the changes, only an estimate. However, we found that most CRs are accepted, and the acceptance rate can have impact on the project plans in terms of decreasing planning precision.

RQ4 (phase): Most CRs are issued pre-delivery, and especially in the short time right after requirement baseline. CRs are issued both before and after implementation of requirements.

The study gives insight into evolution as shown in Figure 1: Quality attributes and functionality are iteratively improved between releases reflecting in the number of CRs to modify solutions, in addition to corrective maintenance. Each release also undergoes

changes mostly in form of new or modified requirements that are adaptive, or functional. It also shows that evolution of earlier releases has stopped as expected. The organization should notice the increased number and acceptance rate of CRs, which require extra resources to handle, and implement these.

The study's contribution is in the empirical evaluation of the intention (categories and goals), origin of changes, and of the distribution between functional and non-functional (quality) requirements/attributes in a large-scale project over time. It also extends the concept of software change to the development and evolution phases when the system is developed incrementally and iteratively.

We also have data on the original requirements in each release, and plan to analyze these to increase our understanding on software change as reflected in requirement evolution between releases.

Acknowledgements

The work was done in the context of INCO (INcremental and COmponent-based Software Development), a Norwegian R&D project in 2001-2004 [15], and as part of the first author's PhD study. H. Schwarz and O.M. Killi gathered the first set of CRs during their work on their joint master thesis in spring 2003. We thank them for the effort. We also thank Ericsson in Grimstad for the opportunity to perform this study.

References

[1] Algestam, H., Offesson, M., Lundberg, L.: Using Components to Increase Maintainability in a Large Telecommunication System. *Proc. 9th International Asia-Pacific Software Engineering Conference* (APSEC'02), 2002, pp. 65-73.

[2] Baldassarre, M.T., Bianchi, A., Caivano, D., Visaggio, C.A., Stefanizzi, M.: Towards a Maintenance Process that Reduces Software Quality Degradation Thanks to Full Reuse. *Proc. 8th IEEE Workshop on Empirical Studies of Software Maintenance* (WESS'02), 2002, 5 p.

[3] Basili, V.R: Viewing Maintenance as Reuse-Oriented Software Development. *IEEE Software*, 7(1): 19-25, Jan. 1990.

[4] Bennett, K.H., Rajlich, V.: Software Maintenance and Evolution: a Roadmap. In *ICSE'2000* - Future of Software Engineering, Limerick, 2000, pp. 73-87.

[5] Damian, D., Chisan, J., Vaidyanathasamy, L., Pal, Y.: An Industrial Case Study of the Impact of Requirements Engineering on Downstream Development. *Proc. IEEE International Symposium on Empirical Software Engineering* (ISESE'03), 2003, pp. 40-49.

[6] Jørgensen, M.: The Quality of Questionnaire Based Software Maintenance Studies*, ACM SIGSOFT* - Software Engineering Notes, 1995, 20(1): 71-73.

[7] Lehman, M.M.: Laws of Software Evolution Revisited. In Carlo Montangero (Ed.): Proc. European Workshop on Software Process Technology (EWSPT96), *Springer LNCS 1149,* 1996, pp. 108-124.

[8] Lientz, B.P., Swanson, E.B., Tompkins, G.E.: Characteristics of Application Software Maintenance. *Communications of the ACM*, 21(6): 466-471, June 1978.

[9] Malaiya, Y., Denton, J.: Requirements Volatility and Defect Density. *Proc. 10th IEEE International Symposium on Software Reliability Engineering* (ISSRE'99), 1999, pp. 285-294.

[10] Mockus, A., Votta, L.G.: Identifying Reasons for Software Changes Using Historical Databases. *Proc. IEEE Int. Conference on Software Maintenance* (ICSM'00), 2000, pp. 120-130.

[11] Mohagheghi, P., Conradi, R.: Using Empirical Studies to Assess Software Development Approaches and Measurement Programs. *Proc. 2nd Workshop in Workshop Series on Empirical Software Engineering* (WSESE'03), 2003, pp. 65-76.

[12] Mohagheghi P., Conradi R., Killi, O.M., Schwarz, H.: An Empirical Study of Software Reuse vs. Defect-Density and Stability, Accepted for ICSE'2004, 10 p.

[13] Rational Inc. www.rational.com

[14] Schach, S.R., Jin, B., Yu, L., Heller, G.Z., Offutt, J.: Determining the Distribution of Maintenance Categories: Survey versus Measurement. *Empirical Software Engineering: An International Journal*, 8(4): 351-365, Dec. 2003.

[15] INCO project: http://www.ifi.uio.no/~isu/INCO/

[16] Sommerville, I.: *Software Engineering*. 6th Ed., Addison-Wesley, 2001.

[17] Stark, G., Skillicorn, A., Ameele, R.: An Examination of the Effects of Requirement Changes on Software Releases. *CROSSTALK - The Journal of Defence Software Engineering,* Dec. 1998, pp. 11-16.

[18] Zowghi, D., Nurmuliani, N.: A Study of the Impact of Requirements Volatility on Software Project Performance. *Proc. 9th International Asia-Pacific Software Engineering Conference* (APSEC'02), 2002, pp. 3-11.