An investigation of change effort in two evolving software systems

Technical report 11/2008, Simula Research Laboratory

Hans Christian Benestad, Bente Anda, Erik Arisholm

Abstract Making changes to software systems can prove costly and it remains a challenge to understand the factors that affect the costs of software evolution. This study sought to identify such factors by investigating the effort expended by developers to perform 336 change tasks in two different software organizations. We quantitatively analyzed data from version control systems and change trackers to identify factors that correlated with change effort. In-depth interviews with the developers about a subset of the change tasks further refined the analysis. Two central quantitative results found that volatility of requirements and dispersion of changed code consistently correlated with change effort. The analysis of the qualitative interviews pointed to two important, underlying cost drivers: Difficulties in anticipating side effects of changes and difficulties in comprehending dispersed code. This study demonstrates a novel method for combining qualitative and quantitative analysis to assess cost drivers of software evolution. Given our findings, we propose improvements to design practices and development tools to reduce the costs.

1 Introduction

Software systems must evolve to adapt to continuously changing environments [1]. With a greater understanding of the cost of software evolution, technologies and practices could be improved to act against typical cost drivers. Development organizations could also make more targeted process improvements and predict cost more accurately in their specific context. Researchers have taken a number of different approaches toward understanding the cost of software evolution. One class of studies has investigated project factors, such as maintainer skills, the size of teams, development practices, and documentation practices, [2-5]. Other studies have examined how system factors such as structural attributes of source code, relate to the ease of changing software [6-8]. A third class of studies has focused on human factors and has probed the individual cognitive processes involved when developers attempt to comprehend and change software [9].

This case study assumes that software evolution consists of change tasks that developers perform to resolve change requests, and that change effort, i.e. the effort expended by developers to perform these tasks, is a meaningful measure of software evolution cost. Thus, by identifying the drivers of change effort we can better understand the cost of software evolution.

Change effort might be affected by factors such as volatility of change requirements, types of change, developer experience, task size and complexity, and structural attributes of the system. An important element of the study design was to propose cost drivers on the basis of a systematic literature review of change-based studies. With this basis, it was possible to separate between i) a confirmatory analysis to test the effect of factors shown to be important in earlier change-based studies and ii) an explorative analysis that identifies factors that best explained change effort in the data at hand. This is also the first study we are aware of that combines quantitative and qualitative analysis of change tasks in a systematic manner. The purpose was to paint a rich picture of factors that are involved when developers spend effort to perform change tasks. Ultimately, our goal is to aggregate evidence from change-based studies into theories of software evolution.

Quantitative data for this study was retrieved from version control systems and change trackers of two independent projects over periods of 6 and 11 months, respectively. The developers recorded the effort to perform change tasks and we used this as a response variable in quantitative models. Qualitative data was collected through semi-structured interviews focusing on the changes that the interviewees had recently made.

The main contributions of this paper are threefold: First, from a *local perspective* the study results can be used to improve the practices in the investigated projects. For example, the study identifies specific factors that were insufficiently accounted for when the projects estimated change effort. Second, from the *software engineering perspective* the study clarifies factors that

Authors
Institution, Address
e-mail:

drive cost of software evolution. For example, the study identifies commonly used design practices that had an undesirable effect on change effort. Third, from the *empirical software engineering perspective* the paper demonstrates a methodology of qualitative and quantitative analysis of software changes to assess factors that affect the cost of software evolution.

The remainder of this paper is organized as follows: Section 2 describes the design of the study, and includes a measurement model based on a literature review of empirical studies of software change. Sections 3 and 4 provide the results from the quantitative analysis, while Section 5 provides the results from the qualitative analysis. Section 6 summarizes the results for the different parts of the analysis and discusses the consequences of the results. Section 7 discusses threats to validity, and Section 8 concludes.

2 Design of the study

2.1 Research question

The study addresses the following research question:

Which factors associated with change tasks correlate with and affect change effort?

As implied by this question, the individual change task is the unit of analysis for this study. Change trackers and version control systems were the source of quantitative data to capture factors that vary across change tasks, such as volatility of change requirements, type of change, experience of the developers who performed the change, and size of the change. Data on change effort was retrieved from the same source.

We use regression analysis to identify factors that best explain variations in change effort. This is the basis for the objective and quantifiable results. However, such analysis does not reveal all factors involved when developers spend effort to perform change tasks. We interviewed developers about recent change tasks to identify factors that were not captured by the quantitative data. Also, because statistical regression analyzes correlations, we expected the interview data to reveal more about the involved causal relationships. For example, regression analysis may show that corrective changes are more expensive than non-corrective changes, but provides little insight into the root causes for this result. Such insight can be found by interviewing developers about how corrective and non-corrective changes were made.

Factors such as the size and type of the system, commercial terms and collaboration model are stable across change tasks, and constitute the context for the study results. Generalization of the results to other contexts is discussed in Section 2.3.

2.2 Case study procedures

Mutual commitment for collaboration was established in 2006 with two development organizations that fulfilled the requirements for the study. Developer interviews were conducted over a period of six months until July 2007, at which point quantitative data was retrieved from change trackers and version control systems in the two organizations. More details about data collection are provided in Section 2.4.

An important preparatory step for the study was a systematic literature review of existing change-based studies [10]. The review identified factors that might influence change effort, and possible quantitative measures to capture these factors. The measurement model described in Section 2.5 summarizes these findings, and defines the specific measures collected for the quantitative analysis.

The quantitative analysis proceeded in two steps: First, we conducted a confirmatory, evidence-driven analysis to test whether a small set of pre-selected measures contributed to change effort in statistical regression models. These measures captured cost factors important in earlier change-based studies. Second, in the data-driven analysis, a wider set of factors and measures were used as input to statistical procedures designed to identify the models that best explained variations in change effort. Section 2.6 describes the specifics in how quantitative data was analyzed.

Roughly once a month, we interviewed the developers about recent change tasks and any circumstances that had made the task easier or more difficult. The interviews aimed to identify additional or more fundamental cost factors than those identified by the quantitative analysis. To achieve this goal, the analysis focused on the changes that had required considerably more or less effort than predicted from the regression models. Section 2.7 describes the procedures that we followed to collect and analyze qualitative data.

The partial evidence from the different parts of the analysis were then compared and integrated into a set of joint results. These results constitute the basis for discussing consequences from the three perspectives that were mentioned in the introduction.

Fig. 1 summarizes the case study procedures. The analysis was based on quantitative and qualitative field data from two software projects, and on proposals generated from existing empirical evidence. The results from each part of the analysis were summarized to strengthen and expand this empirical evidence. With this design, we move towards a theory on software change effort that would be valuable both for researchers and practitioners within software engineering.

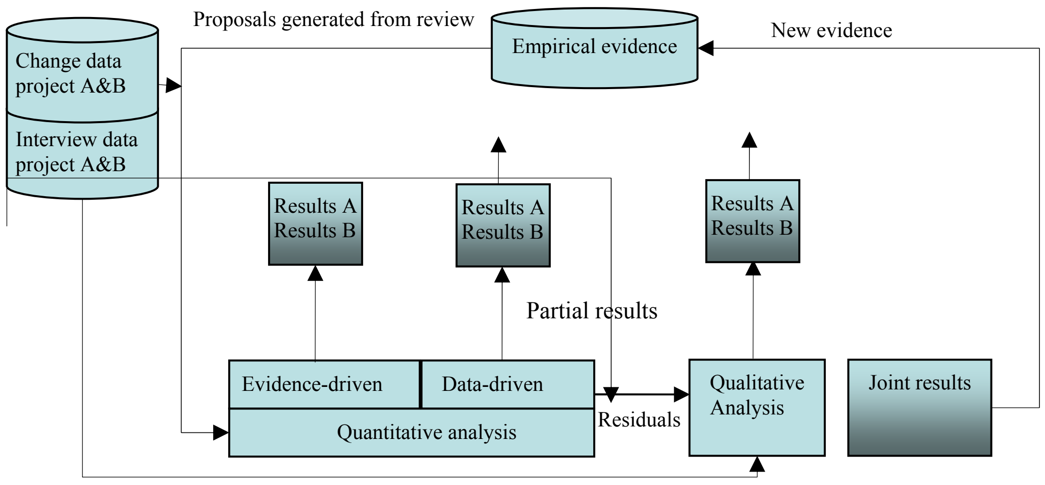

Fig. 1 Case study procedures: Solid lines represent flow of information

2.3 Generalization of case study results

The case study paradigm is appropriate when investigating complex phenomena, especially when it is difficult to separate the investigated factors from their context [11]. In software development and software evolution, social and human factors interact with technological characteristics of the software that is developed. We chose the case study method because we wanted to consider the full complexity of factors that could affect change effort in a realistic context.

A main concern with case studies is whether it is possible to generalize study results beyond the immediate study context. Case study methodologists recommend that studies are designed to build or test *theories*. Theories can then explain, predict and manage the investigated phenomenon in some future situation, and are therefore useful to be able to generalize from case studies. Because we are not aware of theories that are directly relevant to the research question, the proposals for this study were based on a systematic review of relevant empirical evidence. In other words, the systematic review of empirical evidence takes the place of theories in this study.

In particular, the evidence-driven analysis was essential for the generalizability of this study because it is designed to confirm, refute or modify the current empirically based knowledge about factors that correlate with or affect change effort. The role of the data-driven analysis was to discover additional relationships within the investigated projects, and to generate proposals for further confirmatory studies.

The qualitative analysis aimed at refining the quantitative results. For example, while regression analysis could show that more effort is expended when a particular programming language was used, interviews could reveal that developers used this programming language for a particular type of tasks, say, to interface with hardware. This is important in order to make appropriate use of the study results in other contexts, and hence for generalizability.

The results of this study are inevitably under the influence of context factors pertaining to the investigated development organizations. Understanding these factors makes it easier to judge the applicability of the results in a new context. By replicating the study across two development organizations, and comparing the results and the organizations, we were able to evaluate some of these context factors.

2.4 Case selection and data collection

Gaining access to software engineering data of the type required by this study is not straightforward. We approached medium and large-sized software development organizations in the geographic area of our research group during 2006, using procedures that conformed to those described in [12]. We required the participants to grant access to the planned sources for quantitative and qualitative data, to use object-oriented programming languages, to have planned development for at least 12 months ahead, and to use a well-defined change process that included some basic data collection procedures. In particular, when the developers committed code changes to the version control system, they included an identifier of the associated change request in the log message. For each change, the total effort expended on detailed design, coding, unit testing and integration was recorded. The recruitment phase ended when we made agreements with two projects, henceforth named project A and project B.

Project A develops and maintains a Java-based system that handles the lifecycle of research grants for the Research Council of Norway. A publicly available web interface provides functionality for people in academia and industry to apply for research grants, and to report progress and financial status from ongoing projects [13]. The officials of the Council use a Java client to review the research grant applications and reports. The system integrates with a web publishing system, an archive system, and a proprietary system that manages the research programs. The consultancy company that we cooperated with was subcontracted by the Council annually to make improvements and to add new functionality to the system. Most change requests originated from the users at the Council. Roughly once a month, the development group agreed with user representatives and the product owner on changes to include in the next release. They continued to work closely with the development group during design, coding, test and integration of the changes. For the most part, the contractor was paid per hour of development effort. Defects that were detected after deployment were corrected under a guarantee agreement, at no charge

Project B develops and maintains a Windows PocketPC system written in Java and C++. The system allows passengers who travel with the Norwegian State Railways [14] to purchase tickets on-board, and offers electronic tickets and credit card payment. The system integrates with a back-end accounting system that is shared with other sales channels. The consultancy company that we cooperated with had been subcontracted by the Norwegian State Railways to develop the system. In the period of data collection, improvements, new functionality and corrections were made to the system. The main focus was to support a new electronic ticket system, shared between public transport operators in the geographic area. Most change requests originated from the product owner and user representatives. The members of the development group prioritized and assigned development tasks directly in the change tracker, or as part of short and frequent meetings. New versions of the system were released roughly once a month. For the most part, the contractor was paid per hour of development effort. Some larger changes were performed on a fixed-contract, while defects that were detected after deployment were corrected under a guarantee agreement, at no charge.

Both projects were medium-sized and with extensive change activity. Three to six developers were making code changes to the systems in each of the projects.

Fig. 2 and Fig. 3 illustrate change activity and system size over a period of 30 months. Project A deployed the first version of their system in Q1 2003, while project B deployed the system in Q1 2005. Data was collected over the last 6 and 11 months of the charted period, for project A and B, respectively. The apparent dip in system size for project A around Q3 in 2005 was due to a reorganization of the project, where one subsystem was extracted out and defined as a separately managed project. Also, a major change in the technology platform happened at that time.

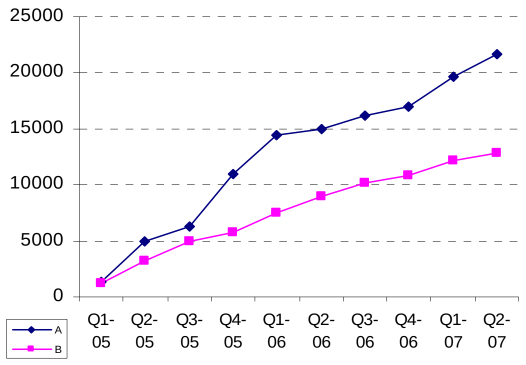

Fig. 2 Accumulated number of commits

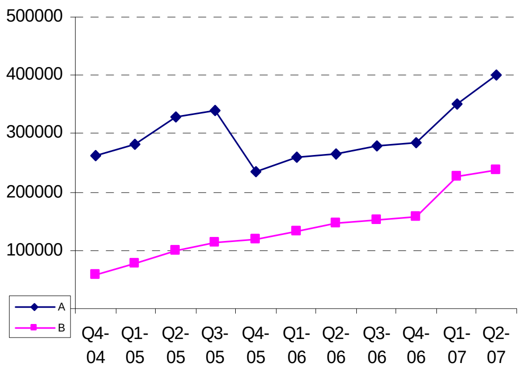

Fig. 3 System size, in lines of code

We developed scripts to retrieve quantitative data from the repositories of the version control systems and change trackers. Raw data was aggregated by MS Access into the change-level measures described in Section 2.5. Qualitative data was collected through a series of interviews with the developers. The interview sessions were aligned with the release rhythm of the projects, i.e. roughly once a month. Table 1 provides key information about collected data.

Table 1 Key information about collected data

	Project A	Project B
Number of analyzed changes	136	200
Total effort of analyzed changes	1425 hours	1115 hours
Changes discussed in interviews	120	65
Period for data collection	Aug 2006 – Jul 2007	Jan 2007-Jul 2007
Version control system	IBM Rational Clearcase LT [15]	CVS [16]
Change tracker	Jira [17]	Jira [17]
Total duration of interviews	20 hours	10 hours
Total time charged for data collection	18 hours	14 hours

The companies charged their normal hourly rate for the time they used on interviews and to record effort data. This agreement was made in order to increase their commitment to provide the required data.

Prior to the analysis, four and six data points were removed from project A and B, respectively, because they corresponded to continuously ongoing maintenance activities, rather than independent and cohesive tasks.

2.5 Measurement model

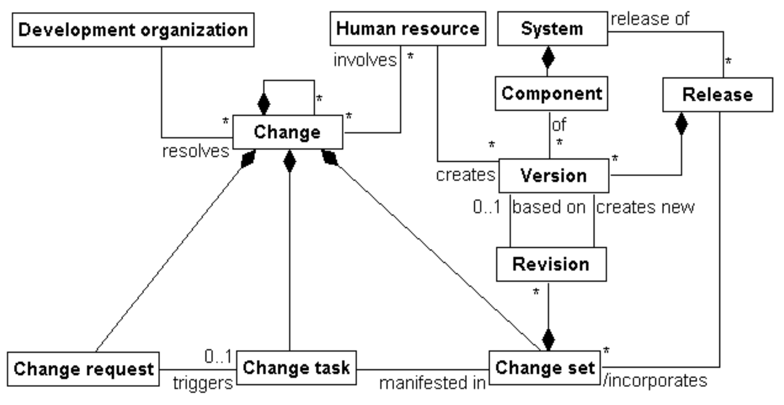

Fig. 4 Key terms and concepts

This study's perspective is that software evolution is organized around the *change task*. A conceptual model for change-based studies is given in Fig. 4. A *change task* is a cohesive and self-contained unit of work that is triggered by a *change request*. In the investigated projects, a change task consists of detailed design, coding, unit testing and integration. A change task is manifested in a corresponding *change set*. A change set consists of *revisions*, each of which creates a new *version* of a *component* of the *system*. The new version can be based on a pre-existing version of the component, or it can be the first version of an entirely new component.

A system is deployed to its users through *releases*. A release is built from particular versions of the components of the system. A release can also be described by the change sets or corresponding change requests that it incorporates. The term *change* aggregates the change task, the originating change request, and the resulting change set. Changes involve *human resources*, and are managed and resolved by the *development organization*. Changes can be hierarchical, because large changes may be broken down into smaller changes that are more manageable for the development organizations.

The measures used as explanatory variables in quantitative models of change effort were intended to capture factors pertaining to the entities of the model shown in Fig. 4. A summary of the relationships between entities, factors and measures is provided in Table 2. In the following, we describe the rationale and empirical foundation for the proposals that certain factors will affect change effort. For each factor, we select one primary measure and zero or more alternative measures. The primary measures are used as explanatory variables in models that are built in the evidence-driven analysis. These models are a reference point allowing us to assess the added value of the data-driven analysis, where we build optimized, project-specific models using all the described measures as candidate variables. We preferred primary measures that were likely to be robust to variations in measurement context, that have been used and validated in previous empirical studies, and that were measurable or assessable at an early stage in the change cycle. Measures are written in *italics*, while primary measures are marked with an additional asterisk (*).

Table 2 Summary of measures

Entity	Factor	Measure	Explanation of measure
Change task	Change effort	*ceffort*	Time expended to design, code, test, and integrate change, tracked by developers Used as response variable in the study.
Change request	Requirement volatility	*crTracks**	-Change tracks for CR before first check-in
		crWords	-Words in CR before first check-in
		crInitWords	-Words in original CR
		crWait	-Calendar time before first check-in
	Change type	*isCorrective**	-Classification + text scanning
Change set	Change set size	*components**	-Changed components
		addLoc	-Measures collected by
		chLoc	parsing side-by-side
		delLoc	output (-y)
		newLoc	of unix/linux *diff*
		segments	-diff –y v2 v1 | cut –c65 | tr –d '\n' | wc –w
	Change set complexity	*addCC*	Parse output of *diff* to measure the number of
		delCC	structural elements added and deleted.
		addRefs	Measures control-flow statements and reference
		delRefs	symbols (. ->)
Component version	Structural attrib.:		
	Size	*avgSize**	-Average/weighted (by *segments*) size of
		cpSize	changed components
	Coupling	*avgRefs*	-Average/weighted (by *segments*) number of
		cpRefs	references to members of imported components
	Control flow	*avgCC*	-Average/weighted (by *segments*) number of
		cpCC	control flow statements
Component	Technological heterogeneity	*filetypes*	-Unique file types that were changed
	Specific technology	*hasCpp (A)*	-Change concerns C++ code
		hasWorkflow (B)	-Change concerns the workflow engine
	Code volatility	*avgRevs*	-Average number of earlier revisions
Human resource and Revision	Change experience	*systExp**	-Avg. previous check-ins by developers
		techExp	-Avg. previous check-ins on same technology
		packExp	-Avg. previous check-ins in same package
		compExp	-Avg. previous check-ins in same components
		devspan	-Number of developers participating in change
Development organization	Project identity	*isA**	1 if change belongs to project A 0 if change belongs to project B

2.5.1 Requirement volatility

Before developers start a change task, they need to comprehend the described requirements, and to analyze the impact of the change by some formal or informal procedure. Modifications or additions that the developers or other stakeholders make to the original change request, the requirement volatility, can indicate uncertainty or other problems in envisioning the change incorporated into the system. Such problems could propagate to the coding phase and affect change effort. In [18], requirement volatility correlated with fault proneness, while in [19], requirement volatility correlated with change effort. A straightforward measure of requirement volatility is the number of modifications to the original change request, as recorded in the change tracker (*crTracks*).* Alternative measures include the number of words in the original change request (*crInitWords*), the number of words in all modifications to the change requests (*crWords*), and the elapsed time from a stakeholder created the change request until a developer started the change task (*crWait*).

2.5.2 Change set size

The *change set size* reflects the differences between the current and preceding versions of changed source components. The intuitive notion that the change set size affects change effort is verified by previous studies [19-22]. Other studies have shown that after controlling for change type or structural complexity of changed components, discussed below, change set size is not necessarily a significant factor [23-25]. A large change set can indicate that a major bulk of new or changed functionality was coded, or that the change request was incompatible with the current design. A coarse-grained measure of change set size is the number of source components that were changed during the change task (*components*).* Finer granularity measures use text difference algorithms [26] to measure the number of lines of code (LOC) that were added (*addLoc*), deleted (*delLoc*) and changed (*chLoc*). Added code in existing components can be differentiated from code in newly created components (*newLoc*).

We selected a coarse-grained measure of change set size because there is evidence that these perform equally well or better than LOC-based measures in models of change effort [21]. LOC counts are less meaningful in technologically heterogeneous environments, and when tools that generate code automatically are used. Furthermore, LOC counts may become high for conceptually trivial changes, such as when program variables or methods are renamed. For estimation of change effort, it is probably easier to estimate the number of components to change than the number of lines of code to change. An alternative, medium-grained measure is to count the number of disjoint places in the existing code where changes were made (*segments*).

2.5.3 Change set complexity

If the structural complexity of the change set is high, e.g., if there are many changes to the control-flow, or many changes in the usage of members of external components, an increase in change effort beyond the effect of change set size could be expected. Except for one study in the authors' research group [27], we are not aware of any studies investigating this effect of change set complexity on change effort. Fluri and Gall showed that measures of edits to the abstract syntax trees of individual components predict ripple effects better than measures of textual differences [28]. We constructed two measures to capture the number of added control-flow statements and added references to members of external components, *addCC* and *addRefs*. Corresponding measures were constructed for deleted control-flow statements and deleted references to members of external components, *delCC* and *delRefs*. Because these are likely to correlate strongly with measures of change set size, and because they are experimental in nature, we only used these measures in the data-driven analysis.

2.5.4 Change type

Changes can be described according to their origin, importance, quality focus, and a number of other criteria. In change-based studies, the *change type* has been important in order to understand change effort [21, 22, 24, 25, 29]. Corrective, adaptive or perfective change types, as suggested by Swanson [30], was the most commonly used classification schema. A recurring result from existing change-based studies is that corrective changes are more time consuming than other types of change, after controlling for change set size [21, 31]. This does not contradict results from studies that have shown that the mean effort for corrective changes is lower than for other change types [29], because corrective changes tend to have smaller change set size [32]. We chose the classification into corrective and non-corrective changes (*isCorrective**) as the primary measure in the analysis. To identify corrective changes, we combined the categorizations performed by the developers with textual search for words like “bug”, “fails” and “crash” (in the native language) in change request descriptions. The latter step was necessary because the projects tended to underuse the category for corrective change.

2.5.5 Structural attributes of changed components

Relevant parts of the system must be understood in order to perform a change task. The structural attributes of these parts may affect developers’ ability to comprehend software code [33, 34]. Many change-based studies have investigated whether the size of changed modules (*avgSize**) correlate with change effort [19, 22, 23, 28, 35]. Arisholm showed that size and certain other structural properties of the changed source components were correlated with change effort [23]. We constructed alternative measures of control flow complexity and coupling in the changed components. The first measure takes the average number of control-flow statements (*avgCC*) in the changed components, while the second measure takes the average number of references to members of imported components, of each changed component (*avgRefs*). Variations of the measures were constructed by weighting the measures by the relative amount of change in each component (*cpSize*, *cpCC* and *cpRefs*), as proposed in [23].

2.5.6 Code volatility

Historical code changes are typically not uniformly distributed over the components of the system. While many components rarely change, some are involved in a large proportion of the change tasks. We propose that the *code volatility* or change proneness will affect change effort, and that changes to change prone components require *less* effort, simply because the developers are more experienced with changing these components. Conversely, changes to infrequently changed components represent unfamiliarity, and may also indicate more fundamental changes. Higher code volatility could also result in *increased* change effort, because frequently changed modules may experience code decay [36]. However, in the investigated projects, components believed to have decayed due to frequent changes were re-factored, and we therefore expected this effect to be limited. The number of historical revisions, averaged over all changed components (*avgRevisions*), captures code volatility of changed components. Several researchers have used volatility of individual components as a predictor of failure proneness, see e.g., [37]. However, we are not aware of studies that have investigated the relationships between code volatility and change effort. Due to this lack of existing empirical evidence we only used this measure in the data-driven analysis.

2.5.7 Technological heterogeneity

Both projects used a number of tools and technologies. *Technological heterogeneity* refers to the number of different technologies involved in a change. Increased technological heterogeneity may increase change effort, because it sets higher demands on developer skills and because it may not be straightforward to integrate technologies. One simple way to measure technological heterogeneity is to count the number of unique file name extensions among the changed components (*filetypes*). We are not aware of studies that have investigated how technological heterogeneity affects change effort. Due to the lack of existing empirical evidence we only used this measure in the data-driven analysis.

2.5.8 Specific technology

Use of a specific technology can affect change effort. For example, Atkins *et al.* showed that when developers used a tool that supported evolution of system variants, change effort was significantly reduced [24]. In project B, functionality interfacing with hardware was written in C++. We propose that changes that involve C++ will be more expensive to change than other code, which was predominantly written in Java. One rationale is that more specialized knowledge is required to develop code that interfaces to hardware. An effect of the lower abstraction level in C++ as compared to Java would work in the same direction. The binary measure *hasCpp* evaluates to true if any of the changed components were written in C++. Project A used a Java-based workflow engine as an important part of the technological basis. Although the project assumed that they benefited from the high abstraction level of this technology, we wanted to investigate whether the changes involving the workflow engine were different with respect to change effort. The binary measure *hasWorkflow* evaluates to true if any of the changed components were based on the technology of the Java-based workflow engine.

2.5.9 Change experience

Experiments have shown that there can be large productivity differences between individual developers [38, 39]. The developer’s ability to perform a change is determined by a complex set of factors that include general mental abilities, education, different categories of experience, and motivation. In a project setting, large individual differences may be masked by compensation effects, e.g., more experienced developers may be assigned to inherently difficult tasks [21, 31]. If the identity of individual developers were used as a nominal measure, we could have accounted for individual differences. This was not possible because we had agreed to avoid analysis that could be perceived as an assessment of the individual developer. Consequently, we resorted to various counts of previous commits to the version control system as indicators of the developers’ experience. This *change experience* can be measured for a given developer at different granularity levels: A basic measure is the total number of previous check-ins by the developer who performed the change (*systExp**). Other measures include the average number of earlier check-ins of the changed components (*compExp*), packages (*packExp*) or technologies (*techExp*). For *packExp*, we counted earlier check-ins to components in the same packages that were changed. For *techExp*, we counted earlier check-ins to components that matched the file extensions of the changed components. If several developers were involved in the change, the averages of the measures were used, weighted by the number of components changed by each developer. Similar measures were used in [40]. In that study, the coarsest-grained measure (*systExp*) significantly affected the response variable capturing failure proneness, while the other measures did not.

2.6 Analysis of quantitative data

This section describes the statistical framework used to build and assess the regression models. The specific procedures for the evidence-driven and the data-driven analysis are provided in Section 3.1 and 4.1, respectively. The statistical packages used were SAS 9.1 to fit regression models, JMP 6 to create decision trees, and R version 2.6.1 to calculate the cross-validated measures of model fit described in Section 2.6.2.

2.6.1 Statistical procedures

Change effort was used as the response variable for all statistical models. The measures discussed in Section 2.5 were used as candidate explanatory variables. The regression model framework was Generalized Linear Models (GLM) with a *gamma* response variable distribution (sometimes called the error structure) and a *log* link-function, see [41]. One reason to assume gamma-distributed responses was that the effort data distribution has a natural lower bound of zero and was right-skewed with a long right tail. This resembles other kinds of gamma-distributed wait-time data, for example data on time-to-death or time-to-failure. A *log* link function ensures that predicted values are always positive, which is appropriate for wait-time data. The size of effect of a specific explanatory variable x_n is assessed by the proportional change in expected change effort that results from a change to x_n. Because a *log* link-function is used, the proportional change in expected change effort becomes:

$$\frac{\text{ceffort}(x_1=C_1..x_{n-1}=C_{n-1}, x_n=C_n+1)}{\text{ceffort}(x_1=C_1..x_{n-1}=C_{n-1}, x_n=C_n)} = \frac{e^{\beta_0+\beta_1 C_1+..+\beta_{n-1}C_{n-1}+\beta_n(C_n+1)}}{e^{\beta_0+\beta_1 C_1+..+\beta_{n-1}C_{n-1}+\beta_n C_n}} = e^{\beta_n}$$

Cross-project models were constructed to identify effects that were present in both projects, and to formally test for project differences. Project-specific models were constructed to identify effects that were particular to each project, and to quantify those effects in each project.

The p-values, sign and magnitude of the coefficients are inspected to interpret the models. The significance level is set to 0.05. This means that for a variable to be assessed as significant, the probability that the variable has no impact must be less than 5%. It is difficult to interpret coefficients when there is a high degree of multicollinearity between the explanatory variables. In the evidence-driven analysis we attempted to reduce multicollinearity by selecting primary measures designed to capture independent factors. In the data-driven analysis, the results from a principal component analysis identified orthogonal factors in the data sets. The actual amount of multicollinearity in the fitted models was measured by the variance inflation factor (VIF).

2.6.2 Measures of model fit

We chose the cross-validated mean and median magnitude of relative error to assess the fit of models. The basis for these measures is the magnitude of relative error (MRE) which is the absolute value of the difference between the actual and the predicted effort, divided by the actual effort. The measures were calculated by *n-fold cross-validation*. With this procedure, the variable subset to be evaluated was fitted in n iterations on n-1 data points. In each iteration, the fitted model was used to predict the last data point. The mean MRE of these predictions forms *MMREcross*, while the median of the values forms *MDMREcross*. The cross-validated measures are more realistic measures of the predictive ability of regression models than those measures that are not based on cross-validated predictions. This was particularly important during the data-driven analysis, where models were selected on the basis of the MMREcross-measure. Because MdMREcross uses the median of the cross-validated MRE values rather than the mean, it is more robust against the influence of outliers.

Another measure to assess model fit is the percentage of data points with an MRE of less than a particular threshold value. *PRED(0.25)* and *PRED(0.50)* measure the percentages of the data points that have a MRE of less than 0.25 and 0.50, respectively.

As a reference point to assess the model performance, we calculated the measures of model fit for the constant model, i.e. the model that uses a constant value as predictor for all data points.

2.7 Collection and analysis of qualitative data

We prepared for the interview sessions by studying data about each change request in the change trackers and version control systems, and attempted to understand how the changed code fulfilled the change requirements. The interview guide is given in Appendix A. In parts 5, 6 and 7 of the interviews, the developers were encouraged to express opinions about phenomena that had affected change effort. The purpose of the other parts was to elicit context information. To help the interviewee recall what had happened during the change task, we made information about each change easily available during the interview sessions.

The changes with the largest magnitude of relative error (MRE) from the data-driven analysis were selected for in depth analysis. An alternative criterion would have been to select changes with the largest error in absolute hours. However, we considered an error of 10 hours to be more interesting if the effort estimated by the model was 2 hours, than if it was 100 hours. We limited the analysis to data points with an MRE of more than 1.3 for underestimated changes and more than 0.5 for overestimated changes. These limits were set somewhat arbitrarily. Note that the *terms underestimated changes* and *overestimated changes* refer to the relationship between the actual change effort and the expected values that were obtained post hoc on the basis of the regression models.

The interviews were transcribed and analyzed in the tool Transana [42], which provides mechanisms to navigate between transcripts and audio data. This feature made it feasible to re-listen to the original voice recordings throughout the analysis. The interviews were coded in two phases. In phase 1, immediately after each interview session, the interviews were transcribed and coded according to a coding scheme that evolved as more data became available. Eventually, 17 categories and 132 codes were used to capture the contents of the interviews, including context and background information. In phase 2, when the quantitative models had been constructed, it was possible to use these to select the changes that required considerably more or less effort than predicted. When only this subset of changes was considered, the number of applied codes was reduced to 82 for the 17 categories. We then narrowed the focus to the categories and codes that suggested a relationship with change effort. Finally, the exact naming and meaning of codes and categories were reconsolidated to make them more straightforward and easier to understand. The coding schema that resulted from this process is described in Section 5.

3 Evidence-driven analysis

3.1 Models fitted in evidence-driven analysis

The data sets from the two projects were concatenated, and cross-project models were constructed to identify effects present in both projects, and to formally test for project differences:

$$\log(\text{ceffort}) = ß_0 + ß_1\text{crTracks} + ß_2\text{components} + ß_3\text{systExp} + ß_4\text{avgSize} + ß_5\text{isCorrective} + ß_6\text{isA} \qquad \text{(M1)}$$

$$\begin{aligned}\log(\text{ceffort}) = {} & ß_0 + ß_1\text{crTracks} + ß_2\text{components} + ß_3\text{systExp} + ß_4\text{avgSize} + ß_5\text{isCorrective} + ß_6\text{crTracks} * \text{isA} + \\ & ß_7\text{components} * \text{isA} + ß_8\text{systExp} * \text{isA} + ß_9\text{avgSize} * \text{isA} + ß_{10}\text{isCorrective} * \text{isA} + ß_{11}\text{isA}\end{aligned} \qquad \text{(M2)}$$

Model 1 includes one explanatory variable for each of the primary measures. It also includes a project indicator (*isA*) allowing for a constant multiplicative between the projects. Model 2 adds interaction terms between the project indicator and each of the primary measures, allowing for different coefficients for each factor in each project.

Furthermore, two project specific models were fitted, one for each of the two data sets:

$$\log(\text{ceffort}) = ß_0 + ß_1\text{crTracks} + ß_2\text{components} + ß_3\text{systExp} + ß_4\text{avgSize} + ß_5\text{isCorrective} \qquad \text{(M3)}$$

The value of the coefficients and the associated significance levels in the project specific models were used to asses the size and statistical significance of effects. The measures of cross-validated model fit were used to assess explained change effort variability. The model fit of the constant models was used as a yardstick for the assessment:

$$\log(\text{ceffort}) = ß_0 + ß_1\text{isA} \qquad \text{(M4)}$$

3.1.1 Results from evidence-driven analysis

Key information about coefficients in the fitted models is provided in Table 3. Significance levels of 0.05, 0.01 and 0.001 are indicated with one, two and three asterisks, respectively.

Table 3 Coefficient values, significance and model fit in evidence-driven analysis

	Cross project constant model (M4)	Cross project w. project indicator (M1)	Cross project w. interactions (M2)	Project A (M3)	Project B (M3)
Intercept (β_0)	9.91***	9.17***	9.30***	9.44***	9.30***
crTracks	.	0.0750**	0.0756**	0.0800*	0.0756**
components	.	0.0976***	0.119***	0.0759***	0.119***
systExp	.	-0.0000389	-0.000177**	0.0000255	-0.000177**
avgSize	.	-0.0000325	-0.0000614	-0.0000108	-0.0000612
isCorrective	.	-0.277*	-0.110	-0.780***	-0.1098
isA	0.63***	0.182	0.142	.	.
*crTracks*isA*	.	.	0.00436	.	.
*components*isA*	.	.	-0.0429+	.	.
*systExp*isA*	.	.	0.000203**	.	.
*avgSize*isA*	.	.	0.0000505	.	.
*isCorrective*isA*	.	.	-0.670*	.	.
MMREcross	3.29	1.52	1.5192	1.86	1.32
MdMREcross	1.43	0.69	0.6786	0.72	0.60
Pred(25)	0.095	0.20	0.23	0.21	0.25
Pred(50)	0.24	0.36	0.40	0.35	0.43

The summary of the constant model M4 shows that the expected change effort was 5.6 hours and 10.5 hours for project A and B, respectively, and that this difference was statistically significant. The summary of the cross project model M1 shows that the measures of requirement volatility and change set size are significant explanatory variables of change effort in the concatenated data set. After accounting for these variables, there was no longer a statistically significant difference between the projects. However, the model M2 shows that project differences exist, because it includes significant interaction terms for system experience (*systExp*isA)* and change type (*isCorrective*isA)*. Furthermore, the summary of the project-specific models M3 shows that the measure of change type was statistically significant in project A only, whereas the measure of system experience was statistically significant in project B only. We use the results from the project-specific models M3 to assess the sizes of effects of the significant explanatory variables:

The number of updates to the change request prior to the coding phase (*crTracks*) had a significant effect on change effort in all models. A 7% increase in change effort could be expected for each additional track in the change tracker. This size of effect was similar in the two projects.

The number of changed *components* had a significant effect on change effort in the models from both projects. When one additional component was changed, a 12.9% and 7% increase in effort could be expected in project A and B, respectively.

In project A, the significant model term for *isCorrective* indicate that corrective changes was expected to require slightly less than half the effort compared to that required by non-corrective changes ($e^{-0.780}$=46%). Note that this estimate was not confounded by a smaller change set size for corrective changes, because such effects are eliminated (held fixed) when assessing the effect sizes of the individual explanatory variables, see Section 2.6.1. In project B, the change type has no significant effect on change effort.

In project B, the total number of commits to the version control system by the involved developers (*systExp*), was significantly related to change effort. Change effort was expected to decrease by 16.2% for every 1000th check-in performed by a developer. In project A, the effect was small and statistically insignificant.

The estimated coefficients for the average size of changed components (*avgSize*) indicate that change effort was slightly lower when large components are changed, but the effects are very small and statistically insignificant.

Plots of actual versus predicted change effort of projects A and B are provided in Fig. 5 and Fig. 6, respectively. The primary measure of cross-validated model fit, MdMREcross, was down from 1.43 for the constant model to between 0.60 and 0.72 for the rest of the models. Hence, the selected measures explain a fair amount of variability. However, a suggested criteria for accepting a model as good is a value of less than 0.25 for MMRE or MdMRE, and higher than 0.75 from Pred(25) [43]. By this standard, the model fit was relatively poor, and justifies the search for additional relationships through the use of the data-driven analysis presented in Section 4.

The variance inflation factor was less than 1.34 for all the coefficients in all models. Hence, multicollinearity was not a threat to the interpretation of the coefficients.

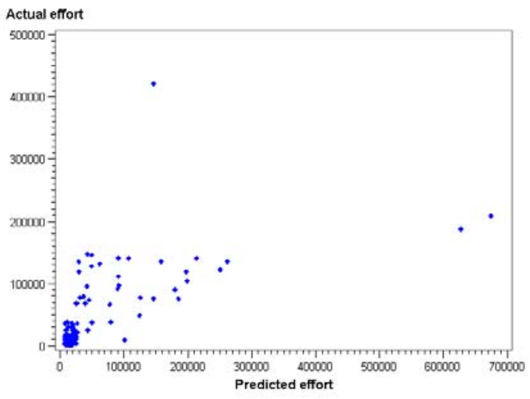

Fig. 5 Predicted vs. actual effort, project A

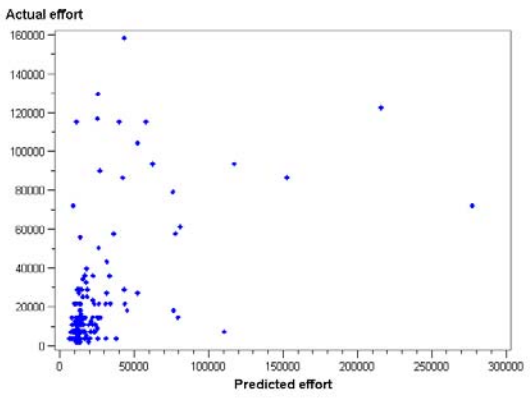

Fig. 6 Predicted vs. actual effort, project B

3.2 Discussion of evidence-driven analysis

In the evidence-driven analysis, we used five pre-selected explanatory variables. The measures of requirement volatility and change set size had a consistent and strong effect on change effort in the models. It is not surprising that the size of the change set was important: The more code that needs to be changed, the more fundamental is the change, and the more effort it takes. It is interesting from a practical perspective that a relatively coarse grained, easily collectable and early assessable measure seems to perform well as a predictor of change effort. It is also possible that *components* captured a particular effect of dispersion that adds to an effect of change set size: Code changes that are dispersed among many components could require more effort than if the same number of lines are changed in fewer components. Both the data-driven analysis and the qualitative analysis investigate this topic in more depth.

The result for requirement volatility is useful to make better effort estimates. The number of comments or tracks in change trackers can be automatically retrieved in an early phase of the change process, and can therefore be a useful predictor for the coding phase. The results also imply that actions that reduce the volatility of requirements can be important to reduce change effort. However, it is difficult to suggest concrete actions without more knowledge about the causes for volatile requirements. This is further investigated in the qualitative analysis.

Corrective changes required less effort than non-corrective changes, although the difference was statistically insignificant in project B. The direction of this effect is opposite to that of earlier studies. A possible explanation is that the tasks and processes involved in corrective vs. non-corrective changes are indeed different, but the direction of the difference is dependent on the situation. A negative coefficient for *isCorrective* indicates that it is relatively easy to correct defects compared to making other types of changes. We consider this to be a favourable situation in projects where it is important to quickly correct defects or where defects are associated with undesirable noise.

The effect of system experience seemed to be quite small, even though *systExp* was statistically significant for project B. One problem with *systExp* as a measure of system experience is that it may be confounded with system decay: The desirable effect of more experienced developers can be counteracted by an effect of system decay, because *systExp* and system decay are inversely related to the underlying factor of time.

We did not measure any significant effect of the size of changed components. There are several possible explanations for this. First, it is not necessarily correct to assume that the class or the file is the natural unit for code comprehension during change tasks. Second, because larger components are more change-prone, developers will have more experience in changing these components. This desirable effect of familiarity may have counteracted an undesirable effect of the size of the components. These issues are further discussed in the qualitative analysis in Section 5.

4 Data-driven analysis

The data-driven analysis presented in this section aims to complement the evidence-driven analysis: It enables us to i) explore relationships that were not originally proposed, ii) assess factors that have a weaker foundation in theory and empirical evidence, and iii) evaluate the predictive power of alternative measures of the same underlying factor.

4.1 Procedures for data-driven analysis

In the data-driven analysis, all the measures from Table 2 were used as candidate variables in the statistical procedures described below. We used:

- Principal component analysis (PCA) to identify subsets of uncorrelated or moderately correlated measures to prevent problems with multicollinearity,
- Cross-validated measures of model fit as the criterion to select the models that best explained change effort.
- Regression trees to identify interaction effects and non-continuous effects.

The goal was to identify the models that explained the most possible change effort variability, under the constraint that each model variable captured relatively orthogonal cost factors.

4.1.1 Variable subset selection based on Principal Component Analysis

The structure of the correlations between the candidate variables was analyzed by principal component analysis (PCA). PCA is used to reduce the dimensionality of wide data sets, and to help in identifying orthogonal factors. Each *principal component* (PC) that results from a PCA is a linear combination of the original variables, constructed so that the first PC explains the maximum of the variance in the data set, while each of the next PC's explains the maximum of the variance that remains, under the constraint that the PC is orthogonal to all the previously constructed PC's. The *loading* of each variable in PC indicates the degree to which it is associated with that PC. In order to interpret a PC, we inspected the variables that loaded higher than 0.5, after the *varimax rotation* [44] had been applied. The results from the analysis are provided in Section 4.2.1.

The results from the PCA were used to construct all possible subsets of candidate variables that contained exactly one variable from each PC. This constraint prevents high multicollinearity in the models, and makes them easier to interpret.

4.1.2 Identify the best models that contain main effects only

For each of the subsets of variables identified by means of the PCA, regression models of change effort were fitted within the described statistical framework, that is, Generalized Linear Models assuming Gamma-distributed outcomes and a *log* link-function. The cross-validated measures of model fit were calculated for the models that only contained significant variables. This requirement helps in interpreting the models, but it was also a pragmatic choice to limit the number of variable subsets that were subject to the cross-validation procedure, which is computationally expensive. The models with the lowest *MMREcross* in the two projects were selected as the best.

4.1.3 Select the best models that include interaction effects or non-continuous effects

The goal of this step was to identify possible interactions between the main effects identified in the previous step, and to discover effects that apply to smaller intervals for values of the explanatory variables. We used a hybrid regression technique that combines the explorative nature of regression trees with the formality of GLM regression, procedures originally proposed in [45]. Regression trees can describe complex interaction effects and non-continuous effects, while still being easy to interpret. Complementary to this, the linear regression framework is suited to identify the overall continuous effects, and to assess the statistical significance of effects.

A regression tree splits the data set at an optimal value for one of the explanatory variables. The split is performed so that the significance of the difference between the two splits is maximized. This step is performed recursively on the splits, until a stop criterion is reached. The stop criterion was that a leaf node should contain not less than 15 data points.

For use in GLM regression, a binary indicator variable was created for each of the leaf nodes in the regression tree. Since this procedure partitions the dataset, every change task had the value 1 for one of the indicator variables, and 0 for the rest. Candidate variable subsets were generated from all possible combinations of the indicator variables and the main effects. The variable subsets that only contained statistically significant variables were retained for n-fold cross-validation. The models with the lowest *MMREcross* from the n-fold cross-validation were selected as the best.

4.2 Results from data-driven analysis

4.2.1 Factors identified by PCA

The summary of results from the principal component analyzes for project A and B are shown in Table 4 and Table 5, respectively. We made the following observations about the match between the conceptual measurement model and the PCA:

- The factors in italics match factors described in Section 2.5. The collected measures for these factors are consistent with the measurement model, and capture five orthogonal factors in the data set: *Change set size, Component version size*, *Requirement volatility*, *Change experience* and *Change type*.
- PC1A and PC2B show that the suggested measures for control-flow and coupling belong to the same principal component as the LOC-based measures of size. The underlying factor captured by all these measures is the size of changed components.
- Likewise, PC1B shows that the suggested measures of change set complexity belong to the same principal component as the LOC-based measures of change set size, in project B.
- PC2A and PC3B contain measures that capture the dispersion of changed code over components, types of components and developers. We label this dimension *change set dispersion*. This dimension captures a factor that is orthogonal to change set size.
- PC3A contains measures of removed code. This principal component captures the *amount of rework*, apparently distinguishable from the concept of change set size in project A.
- In project A, the measure of code volatility belongs to a distinct principal component (PC7A), while in project B, it belongs to the principal component that captures size (PC2B). The latter result indicates that large components are more prone to change, simply due to the effect of size.
- PC6B contains a measure of lines of code in new components, and the change set dispersion. One possible interpretation is that these measures capture the degree of mismatch between the current design and the design required by the change.

These observations are accounted for when the models are interpreted, in Sections 4.3 and 6.

Table 4 Summary of principal component analysis, project A

PC	PC1A	PC2A	PC3A	PC4A	PC5A	PC6A	PC7A	PC8A
Load > 0.5 after varimax rotation	avgSize avgRefs avgCC cpRefs cpCC cpSize	hasWorkflow addCC addRefs newLoc components filetypes devspan	delLoc delCC delRefs crWait	addLoc chLoc segments	crWords crInitWords crTracks	systExp techExp packExp	avgRevs	isCorrective
Entity	*Component version*	Change set	Change set:	*Change set*	*Change request*	*Human resource*	Component version	*Change request*
Factor	*Size*	Dispersion	Rework	*Size*	*Requirement volatility*	*Change experience*	Code volatility	*Change type*

Table 5 Summary of principal component analysis, project B

PC	PC1B	PC2B	PC3B	PC4B	PC5B	PC6B	PC7B
Load	addLoc	avgSize	components	crWords	systExp	newLoc	isCorrective
> 0.5	delLoc	avgRefs	filetypes	crInitWords	techExp	components	
after	chLoc	avgCC	devspan	crTracks			
varimax	segments	avgRevs	packExp	crWait			
rotation	addCC	cpRefs	hasCpp				
	delCC	cpCC					
	addRefs	cpSize					
	delRefs						
Entity	*Change set*	*Component version*	Change set	*Change request*	*Human resource*	Change set	*Change request*
Factor	*Size*	*Size*	Dispersion	*Requirement volatility*	*Change experience*	Design mismatch	*Change type*

4.2.2 Main effects

Candidate variable subsets were generated from all combinations of variables that included at most one variable from each principal component. On the basis of the results in

These observations are accounted for when the models are interpreted, in Sections 4.3 and 6.

Table 4 and Table 5, this meant that 71680 variable subsets were generated for project A and 38880 variable subsets for project B. Following the procedure described in Section 4.1.2, the variable subsets shown in the first and second rows of Table 6 were eventually selected as the best variable subset.

For project B, the best variable subset from the data-driven analysis includes all the primary measures used in the evidence-driven analysis. In addition, the model includes the measure *addCC*, which counts the number of control-flow statements in the change set. This was intended to capture structural complexity of the change set, but results from the PCA showed that *addCC* must be considered to be a size measure in this data set. The size of effect is moderate, as the expected change effort increases by 10% when 10 control-flow statements are added. For the other explanatory variables, the sizes of effects are similar to those described in the evidence-driven analysis.

For project A, only the change type indicator *isCorrective* recurred from the evidence-driven analysis. The model predicts that 40% less effort is required for corrective changes. The measure *crWords* is the number of words in the tracks counted by the primary measure *crTracks*. An increased change effort of 10 percent can be expected when 50 additional words are used in updates of the change request. A strong effect is indicated by the coefficient for *filetypes*, the measure of technological heterogeneity: Change effort is expected to increase by around 30 % when one additional file type is part of the change set. A measure of change set size, *chLoc*, is also significant: An increase of 30 % can be expected when around 50 additional lines of code were changed.

The variance inflation factor was lower than 1.88 for all the coefficients in the two models. This verifies that multicollinearity is not a problem for the interpretability of the coefficients.

4.2.3 Interaction effects

As explained in Section 4.1.3, regression trees were used to explore interaction effects and non-continuous effects. A binary variable was constructed for each of the leaf nodes, and used as candidate variables in GLM regression together with the main effects of the models described in the previous section. For project A, there were four main effects and seven leaf nodes, which give rise to 2048 candidate variable subsets. For project B, there were four main effects and eight leaf nodes leading to 4096 variable subsets.

The variable subset that resulted in the lowest MMREcross was selected for each data set. The third row in Table 6 contains a summary of the selected model for project A. Compared with the model that contained main effects only (row 1), the new model retains three of the four main effects, and adds four interaction rules. The first three of the interaction rules identify the 50 changes that involve only one file type, i.e. technologically homogenous changes. The coefficients for the rules are negative, which means that change effort is lower for these changes, beyond the difference that is explained by the continuous main effect. The second and third rules indicate that the effect of technology homogeneity is weaker when the requirement volatility is higher. Rule 3 indicates that the effect also decreases with large change set size. The fourth rule predicts that 2.6 times more effort is expended for changes involving three or more file types and many changes to existing code.

The last row in Table 6 contains a summary of the model with the lowest MMREcross for project B. Compared with the model that contained main effects only (row 2), a binary rule replaces a continuous effect of *addCC*: If 23 or more control-flow statements are added, then this doubles the expected change effort. This rule applies to 12% of the changes.

Table 6 Coefficient values, significance and model fit in data-driven analysis

Model	Variable	Coefficient	MMRE cross	MdMRE cross	Pred (25)	Pred (50)
Project A Main effects	Intercept	9.06***	1.52	0.63	0.23	0.40
	crWords	0.00187**				
	filetypes	0.279***				
	chLoc	0.005111**				
	isCorrective	-0.503*				
Project B Main effects	Intercept	9.06***	1.12	0.60	0.24	0.42
	crTracks	0.0879***				
	addCC	0.00949**				
	components	0.1027***				
	systExp	-0.000161**				
Project A With interaction terms	Intercept	9.64***	1.37	0.57	0.24	0.46
	crWords	0.00109*				
	filetypes	0.178***				
	isCorrective	-0.376*				
	filetypes=1 & *crWords*<24	-1.145***				
	filetypes=1 & *crWords*>=24 & *chLoc* < 2	-0.831***				
	filetypes=1 & *crWords*>=24 & *chLoc*>=2	-0.653**				
	filetypes>=3 & *chLoc*>= 48	0.963***				
Project B With interaction terms	Intercept	9.15***	1.12	0.62	0.22	0.40
	crTracks	0.0839***				
	components	0.0798***				
	systExp	-0.000153**				
	addCC>=23	0.7877**				

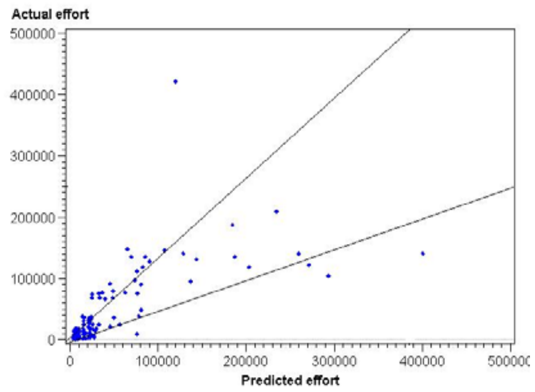

Fig. 7 Predicted vs. actual effort, project A

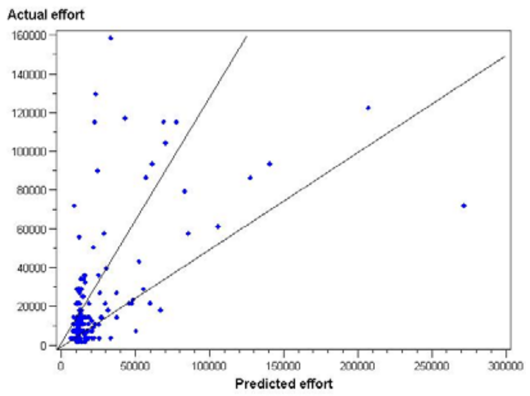

Fig. 8 Predicted vs. actual effort, project B

When regression tree rules were added for project A, the cross-validated model fit substantially improved. For project B, the model fit was almost identical with and without rules from the regression tree.

4.3 Discussion of data-driven analysis

The results from the data-driven analysis complement the results from the evidence-driven analysis. In the model for project B, LOC-based size, as well as change set dispersion, had significant effects on change effort. The PCA showed that these measures captured different factors in the data set. A possible explanation is that change set dispersion affected change effort, beyond the effect of LOC-based size.

Technological heterogeneity can also be an important predictor of change effort. In particular, less effort was expended when only one technology was involved in the change. This result can be useful to improve change effort estimates.

The size and complexity of changed components did not have a significant effect on change effort in the models. At face value, this is a surprising observation because it violates fundamental assumptions about relationships between software design and the ease with which the software can be maintained and evolved. One explanation is that the systems overall had good and modular designs, with only minor (in terms of their effect on change effort) size and complexity differences between components. Another explanation is that the chosen measures do not capture the most important (in terms of their effect on change effort) structural attributes of the source code. This issue is further discussed in the qualitative analysis in Section 5.2.

For project A, the data-driven analysis resulted in models that had better model fit than the models from the evidence-based analysis. In particular, the improved model fit can be contributed to the introduction of a measure of technological heterogeneity. For project B, the model fit did not improve. In this case, the primary measures seemed to capture the important factors available from the data at hand. For both projects, the regression trees proved to be well suited to simultaneously describe non-continuous effects and interaction effects. The total amount of explained change effort variability was moderate. This shows that there were other important factors at play than those captured by quantitative measures. The plots in Fig. 7 and Fig. 8 show the MRE boundaries for overestimated and underestimated changes. The changes that fell outside the area formed by these lines received particular attention during the qualitative analysis. In total, 32 underestimated changes and 16 overestimated changes (those with MRE limits of 0.5 for overestimated changes and 1.3 for underestimated changes, see Fig. 7 and Fig. 8) were analyzed in depth.

5 Results from the qualitative analysis

Table 7 provides a summary of the results from the qualitative analysis of 44 of the 48 selected changes. Four changes were excluded from the analysis because the interviews showed that code changes had not been properly tracked. In other words, the quantitative models identified data points with large measurement error.

The three first columns in Table 7 define the coding schema that resulted from the coding process. Each code captures a factor that was perceived by the interviewees to drive or save effort. For example, *T0* could drive effort if the developer was unfamiliar with a relevant technology, and save effort if the developer had particularly good knowledge about the technology. The rightmost column shows the number of times a code was used in underestimated and overestimated changes, respectively.

Table 7 Summary of factors from qualitative analysis

Category	Code	Description of code	Occurrences in underestimated/over estimated changes
Understanding change requirements	R1	CR clarification was needed/not needed	9/2
Identifying and understanding relevant source code	U1	It was difficult/easy to understand the relevant source code	7/1
	U2	It was difficult/easy to identify the relevant system states	3/3
	U3	The developer was unfamiliar/familiar with relevant source code	3/2
Learning relevant technologies and resolving technology issues	T0	Developer was unfamiliar/familiar with the relevant technology	3/0
	T1	The features of the technology did not/did suite the task	1/2
	T2	Technology had/did not have defects that affected the task	4/0
	T3	Technology had limited/good debugging support	5/0
Designing and applying changes to source code	D1	Change required deep/shallow understanding of user scenario	0/9
	D2	The needed mechanisms were not/were in place	13/2
	D3	Changes were made to many/very few parts of the code	0/8
Verifying the change	V1	It was necessary/not necessary to establish test conditions	2/1
Cause of change	C1	Error by omission – failed to handle a system state	11/5

(analyzed for all changes)	C2	Error by commission – erroneous handling of a system state	1/3
	C3	Improve existing functionality – within current system scope	4/9
	C4	Planned expansion of functionality – extend the system scope	6/5

Many of the codes and categories coincide with concepts studied within the field of software comprehension. For example, Von Mayrhauser and Vans proposed and evaluated an integrated model of software comprehension that addressed the comprehension processes of developers who perform change tasks within large scale software maintenance [46]. They suggested lists of activities involved in change tasks that largely conform to the developed categories. One discrepancy is that, in our case, a separate category was justified for technology properties. Also, the design activity was difficult to distinguish from the coding activity; hence we used a common category. We chose to use a common coding schema for all types of changes, and let the cause of change be part of the coding schema.

In the following subsections, we discuss factors that affected effort for the analyzed change tasks. The analysis applies to both projects, except when project particularities are mentioned specifically. The tables in Appendix B provide details about each of the analyzed changes

5.1 Understanding change requirements

R1. For nine of the underestimated changes, the developers mentioned that the need to clarify requirements resulted in increased change effort. For two of the overestimated changes, they mentioned that a concise and complete specification made it easier to perform the change. This supports the results from quantitative analysis, which showed a consistent relationship between the amount of updates to the original change request, and change effort. For the nine underestimated changes, the requirement clarifications were only partially documented in the change tracker. This explains the large residuals for these changes. The need to clarify requirements occurred more frequently in project A than in project B. However, six out of nine underestimated changes for project B were fixes of errors due to missed requirements, see Section 5.6. Hence, incomplete requirements had an undesirable effect in both projects.

In some cases, the developers said that the user representatives deliberately failed to provide complete specifications, in particular for changes that concerned the look and feel of the user interface. However, the strongest effect on effort occurred when unanticipated side effects of a change needed to be clarified during detailed design and coding. In most cases, this meant that existing functionality was somehow impacted by the change, but that the developer was uncertain how to deal with these impacts. In general, it is not obvious that updates to requirements should be avoided. However, in the investigated projects, updates to change requests were indeed associated with higher change effort. We conclude that practices that help to identify side effects of change requirements are likely to have a positive effect in both projects.

5.2 Identifying and understanding relevant source code

Time expended by developers to comprehend code that is relevant for a change can constitute a substantial share of the total change effort. Koenemann and Robertson suggested that the comprehension process involves code of direct, intermediate and strategic relevance [47]. Directly relevant is code that has to be modified. Code that is perceived to interact with directly relevant code has intermediate relevance. Strategic code acts as a pointer towards other relevant parts of the code. These categories conform well to the descriptions provided by the developers in this study: Some code was inspected in order to identify the code that was relevant for the change, i.e. strategic code was comprehended to locate intermediate code. Then, from the intermediate code, the direct locations to make code changes were identified.

U1: Typically, the change requests were described by referencing a *user scenario*, i.e. a sequence of interactions between the user and the system, and by requesting a change to that scenario. For seven of the underestimated changes, the developers expressed that considerable time was spent understanding relevant, *intermediate* code when it was dispersed among many files. In other words, the qualitative data shows that when code involved in the changed user scenario was dispersed over many components, that user scenario was difficult to change.

The dispersion of changed code had a strong and consistent effect on change effort in the quantitative models. It is possible that the time developers spend to *comprehend dispersed code* is a fundamental factor that in many cases explains the apparent effect of *making dispersed changes*. When intermediate code for a change is dispersed, it is likely that direct code changes are dispersed. We did not possess the data required to confirm such a correlation.

The effort involved in comprehending code along the lines of user scenarios can also explain why the measures of structural attributes of changed components did not have an effect on change effort in the quantitative models. First, only directly affected components were captured by these measures, even though the structural attributes of intermediate code were likely to be important. Second, the measures capture the structural attributes of files and classes rather than of user scenarios. This suggests that it would be more useful to collect measures of structural attributes along the execution path of the changed user scenarios. These measures could be based on models such as UML sequence diagrams, which would also aid in comprehension [48], or dynamic code measurement (e.g., by executing each user scenario), as proposed in [49].

U2: For three of the underestimated changes, the developers expressed that it was difficult to identify and understand the system states relevant to the change task. One developer stated: "All the states that need to be handled in the GUI make the code mind-blowing." This statement indicates that the perceived code complexity is caused by a complex underlying state model. It also suggests that in order to understand the code from the functional view discussed above, it is a prerequisite that the underlying state model is understood. An obvious proposal is to provide aid that makes it easier to understand the most complex underlying state models, e.g., by the use of diagramming techniques such as UML state diagrams.

U3: The degree of familiarity with relevant code was said to have affected change effort in five cases. The straightforward explanation is that code familiarity determines the amount of time that is necessary to comprehend code of direct, intermediate and strategic relevance. The quantitative results for change experience showed that relatively little of the variations in change effort can be explained by familiarity with the systems. The qualitative analysis showed that experience was indeed important in both projects, in the few extreme cases when it was either very high or very low.

5.3 Learning relevant technologies and resolving technology issues

T0. Project A used several different technologies. Lack of familiarity with relevant technology was perceived to increase change effort for three of the changes. The measure that was intended to capture the effect of technology experience (*techexp*), was not significant in the quantitative analysis. One possible explanation is that familiarity with the involved technology affected change effort in the relatively few cases where the familiarity was particularly low or high.

T1, T2, T3: The degree of match between the actual and required features of the development tools and technologies was considered important in 12 cases. If the functionality required by the change task was provided out of the box, the technology was considered to save effort. Reversely, if the technology was incompatible with the change task, or had defects, considerable effort was required to create workarounds. Unsatisfactory facilities for debugging were considered to increase change effort in five cases. We conclude that these factors are important to consider when evaluating technologies for use in a development environment.

5.4 Designing and applying changes to source code

D1: Empirical studies have shown that the nature of a given task determines how the comprehension process is carried out [50]. Indeed, the interview data showed that the developers associated a certain degree of superficiality or *shallowness* with a change task. A change was perceived as shallow when the developer assumed that it was not necessary to understand the details of the code involved in the changed user scenario. Typically, shallow changes were performed by textual search in intermediate code to identify the direct code to change. Examples of shallow changes were those that concerned the appearance in the user interface, user messages, logging behaviour and simple refactoring. Deep changes, on the other hand, required full comprehension of the code involved in the changed user scenario. The comprehension activities described in the previous section are therefore primarily relevant for deep changes.

D2: We use the term *mechanism* for code that implements a solution to a recurring need in the system. Typically, formalized design patterns [51] can be used directly or as part of a mechanism. In the investigated projects, examples of mechanisms are handling of runtime exceptions and transfer of data between the physical and logical layers of the system. In 13 cases, the change was perceived to be particularly challenging because a required mechanism had to be constructed as part of the change. According to the developers, creating these mechanisms was challenging for two reasons: First, the mechanism had to be carefully designed for reusability. Second, when the purpose of mechanisms was to hide peculiarities of specific technologies, these needed to be well understood by the developer of the mechanism.

D3: For eight of the overestimated changes, the developers expressed that the change was easy to perform because it was concentrated in one or few parts in the code. This observation supports the results for *change set dispersion* from the quantitative analysis, and suggests a particularly strong effect for the most localized changes. However, this explanation is contradicted by data from 50 other change tasks that affected only one segment of the code without resulting in particularly low change effort. An alternative explanation is that the developers *perceived* the change to be particularly local because the code of intermediate relevance was not dispersed among many components, as elaborated in Section 5.2

5.5 Verifying the change

V1: The effort expended to test the developers' own code changes was discussed in the interviews. For a large majority of the changes, the developers expressed that it was quite easy to verify that the change was correctly coded. In two cases, verification was perceived to be difficult because the change task affected time-dependent behaviour that had to be simulated in the test environment. In project A, some extra time was needed when it was necessary to generate and execute the system on the target mobile platform. In project B, extra time was needed when the technology necessitated deployment on a dedicated test server.

5.6 Cause of change

The cause of each change, i.e. the events that triggered the change request, was discussed with the developer assigned to the particular change task. Based on this, we classified all changes according to the codes shown in the last row of Table 7. In order to better understand the results for change type from the quantitative analysis, we measured the agreement between the automated classification into change types, and the classification from qualitative analysis. Sufficient data was available for 87 and 61 changes, for project A and B, respectively. When mapping C1 and C2 to corrective change, and C3 and C4 to non-corrective change, the agreement was good (Cohen's kappa=0.64) for project A, but less than what could be expected by pure chance (Cohen's kappa=-0.038) for project B. This result shows that the automated classification for project B did not appropriately reflect real differences in change type, which can explain why there was no effect of change type in the quantitative models. From the qualitative analysis of project B, it can be seen that six out of nine of the underestimated changes were fixes of *error by omission*. A typical reason for such an error was not recognizing a side effect of a change. We conclude that for project B, fixes of errors by omission were associated with underestimated changes. In line with the conclusion in Section 5.1, we recommend practices that help to identify side effects of change requirements, because they are likely to reduce occurrences of errors by omission.

6 Joint results and discussion

On the basis of a systematic literature review of earlier change-based studies, we proposed a small set of factors and straightforward measures that could explain variations in change effort. The evidence-driven analysis largely confirmed the proposals, and strengthens the evidence that these factors are not spurious, but present over time and across software development contexts. The data-driven analysis confirmed the results from the evidence-driven analysis, and provided evidence of additional, more project-specific effects.

The explained variability in the quantitative models was relatively poor by standards that have been suggested for prediction models [43]. The qualitative analysis identified cost drivers that had not been captured by the quantitative measures, by focusing on change tasks that corresponded to large model residuals. The concrete results of the analyses were:

- Requirement volatility, measured by updates in the change tracker, consistently contributed to change effort in the quantitative models. The qualitative analysis showed that when requirement volatility was due to difficulties in anticipating side effects of a change, the effect was particularly large.
- Change set dispersion, measured by the number of changed components or types of changed components, consistently contributed to change effort in the quantitative models. The qualitative analysis suggested that the effort expended by developers in comprehending highly dispersed code was an important underlying cost driver.
- Overall, measures of change set dispersion were better predictors of change effort than were more fine-grained (e.g., LOC-based) measures of change set size.

- In project A, corrective changes required more effort than non-corrective changes, after accounting for other factors. No significant difference was found for project B. The qualitative analysis showed that a sub-class of corrective changes (fixes of errors by omission) in many cases required extra effort.
- A statistically significant, but small effect of developers' experience was identified in project B. The qualitative analysis showed that familiarity with the changed functional and technological areas was indeed important in both projects, in a few more extreme cases when the familiarity was either very high or very low. This effect of experience was not appropriately captured by the quantitative measures.
- Structural attributes of changed components did not have a significant effect on change effort in the quantitative models. The qualitative analysis showed that the properties of the code involved in the changed user scenario did affect change effort. In particular, the complexity of the underlying state model of the user scenario was important, as was the dispersion of code that implemented the user scenario.
- The qualitative analysis showed that change effort increased when the relevant tools and technologies had defects, were inadequate for the task, or did not support debugging satisfactorily.
- The qualitative analysis showed that certain properties of the change task, such as the need for *innovation* in the change task, or *shallowness* of the change task were important factors that we had not attempted to capture by the quantitative measures.

In the following, we discuss the consequences of these results from the perspective of software engineering, the local projects, and that of research methods within empirical software engineering.

6.1 Consequences for software engineering

The results from the study have implications for effort estimation of change tasks during software evolution. First, due to the wide prediction intervals implied by the relatively poor model fit obtained in this and similar studies [22, 35], it seems infeasible to build models that are sufficiently accurate to be accepted as a black-box method to estimate the effort expended on individual change tasks. Effort estimates generated by models may still play a role to support projects in planning releases during software evolution, where the primary interest is in the aggregate of change effort estimates. This is because the aggregated prediction interval decreases, measured proportionally to the estimate, as more predictions are aggregated. On the basis of results from this and earlier change-based studies, we recommend that models include measures of requirement volatility, developers' experience, type of change and change set dispersion. As shown in this study, it is feasible to automatically retrieve measures of the first three factors from version control systems and change trackers prior to the coding phase. A coarse grained impact analysis would be necessary to obtain a measure of change set dispersion in this phase.

The results can also be used to help experts improve judgement-based effort estimates. One method is to develop checklists of factors that experts should assess when they make estimates. Projects can either retrieve appropriate measures from version control systems and change trackers, or they can make subjective judgements for each factor.

Regression models of change effort can also be used to investigate the effect of a new practice or technology. A generic setup is to complement the model with a binary variable that indicates whether the practice or technology was used for a particular change [21, 24, 27, 52]. The significance, sign and magnitude of the coefficient for this explanatory variable can then be used to assess the effect of the new practice or technology. The types of measures that we recommend in such models conform to those recommended in [21], with the important addition of a measure of requirement volatility.

A recommended best practice for software design is to distribute responsibility between relatively small, collaborating objects [53]. The study adds to the empirical evidence that delocalized, or dispersed code causes difficulties during program comprehension, see [54] and [55]. A consequence of these findings is that to facilitate comprehension during software evolution, code that is functionally cohesive should be localized rather than dispersed. This concern about comprehension effort should be balanced against other concerns, such as potentials for reuse and constraints set by the physical architecture.

The results of this study strengthen the case for tools that makes it easier to understand code that cross-cuts architectural units, along functional units such as user scenarios. One feature that already exists in some software modeling tools is the ability to perform static analysis of some selected portion of the code (e.g., a method) to generate a dynamic model of that code (e.g., a

UML sequence diagram) [56]. However, the reverse-engineering of sequence diagrams using dynamic analysis of those objects and messages that are involved in a specific user scenario or use case is still at an early stage of research and development [57]. It is, furthermore, not obvious how this approach could be extended to technologically heterogeneous and physically distributed computing environments.

Context factors of the investigated projects may limit those projects to which the above discussion applies. In order to retrieve change-based measures for the purpose of prediction or assessment, a well-defined, tool-supported change process is required. The client collaboration model is likely to affect volatility of requirements and how this factor is managed: Both projects combined a formal collaboration model at the project level with semi-formality at the level of releases, and informality and close collaboration at the level of change tasks. The nature of the change tasks differed between the projects, the tasks in project B being more corrective than those in project A. As discussed in Section 5.6 a subset of the corrective changes required considerably higher change effort than predicted from the models. This indicates that type of evolution may influence change effort for individual change tasks. We believe that the effect of code dispersion is universal to all kinds of software systems, however particular relevance can be expected in technologically heterogeneous environment, such as in project A.

6.2 Consequences for the investigated projects

In project A, effort estimation was a team activity performed on a regular basis as part of release planning. To judge the potentials for more accurate effort estimates, we calculated the accuracy of the current estimation process, on the basis of effort estimates and actual effort for the 107 change tasks where this data was available. The effort estimates were given in units of relative size, see [58], and were scaled according to the factor that minimized MdMRE. The resulting MMRE and MdMRE was 1.47 and 0.54, respectively. Even though these values roughly correspond to the accuracy of the models from the data-driven analysis, we did not recommend replacing judgement-based estimates with model-based estimates, for two reasons. First, change set size or change set dispersion would have to be subjectively assessed to obtain the required input measures. This would likely decrease the model accuracy, and preclude fully automated procedures. Second, the team estimation of change tasks was perceived to be important to share knowledge and build team spirit in the project, and to constitute an initial step of design for a solution to the change request.

An alternative use of the results was to improve effort estimates that are based on developers' judgement, by ensuring that the important factors are assessed by the developers. To assess whether the factors were already accounted for by the developers, we fitted regression models that included the developers' estimate as an explanatory variable. In these models, measures of requirement volatility, change set dispersion and change type became statistically insignificant. This indicates that these factors were already sufficiently accounted for by the subjective estimates. The number of different technologies involved, on the other hand, had a significant effect on actual effort. The model was:

$$\log(\text{ceffort}) = 9.25 + 0.13*\text{relativeEffortEstimate} + 0.14*\text{filetypes}$$

We recommended that the developers put more emphasise on the latter factor when they made effort estimates of change tasks. Due to the results from the qualitative analysis, we also advised the project to be more aware of the effect of particularly strong familiarity or lack of familiarity with code of intermediate and direct relevance.

Project B used similar, judgement-based procedures for estimates of change effort, but we did not have sufficient data to assess the potentials for more accurate estimates. We were therefore only able to advise that all the identified factors should be assessed when the effort for planned change tasks was estimated.

Even though this study identified factors that correlated with and affected change effort, it was not straightforward to identify specific actions in the projects that would mitigate these factors, and hence save costs. However, on the basis of the results, some actions that we believe would have a positive effect are:

- Further improve knowledge sharing between the system stakeholders
- Refactor code where execution paths are dispersed across more components than necessary
- Acquire tools that make it easier to simulate and understand the code involved in user scenarios
- Document the underlying state models in areas where those models are particularly complex

6.3 Consequences for empirical software engineering

The goal of empirical software engineering is to use empirical methods to assess and improve software engineering practices. In the following, the experiences with three central elements of the design of this study are summarized:

Foundation in a systematic review. The use of systematic reviews in software engineering was suggested as an important element of *evidence-driven software engineering* [59], and the method has already gained significant momentum in the empirical software engineering community. The factors and measures that were used in the quantitative analysis were selected on the basis of a systematic literature review of earlier change-based studies. This was particularly important to be able to perform the evidence-driven analysis, which was key to linking this study to results from previous studies. Systematic reviews have a particularly important role when study proposals cannot be derived from established theories. Currently, this is the situation for most topics that are investigated within the empirical software engineering community.

Combined confirmatory and explorative analysis. According to proposed guidelines for empirical studies on software engineering, strong conclusions can only be drawn from confirmatory studies, while explorative studies are important to generate hypothesis and guide further research [60]. We combined confirmatory and exploratory elements: The evidence-driven analysis largely confirmed proposals about factors that affect change effort. The data-driven analysis explored and identified additional factors that can be investigated in future confirmatory studies.

Qualitative analysis to explain large model residuals. Even though the role of qualitative methods in this field has long been recognized, see e.g., [61], empirical researchers have developed and used quantitative methods to a larger extent [62]. Because we used *the individual change* as a common unit of analysis, and *change effort* as the dependent variable, we were able to tightly integrate the quantitative analysis of data from version control systems and change trackers with the qualitative analyses of developer interviews. The qualitative analysis contributed to the joint analysis since it enabled us to:

- Confirm the importance of factors that were not properly captured by quantitative measures. An example of this was the effect of *developer experience*.
- Identify more fundamental factors than those identified by the quantitative analysis. For example, the apparent effect of change set dispersion could be explained by the *dispersion of comprehended code*.
- Identify additional factors not attempted to be captured by the quantitative analysis. An example is the effect of defects and inadequacy of relevant technologies.

This method can also be used to focus the more expensive qualitative analysis on the most interesting data. This is particularly important for practitioners who use lightweight empirical methods to evaluate their own practices such as Postmortem analysis [63] or Agile Retrospectives [64]. We expect that group discussions that are part of such practices would benefit from focusing on the activities that required considerably more or considerably less time than expected from quantitative data.

7 Threats to validity

Construct validity. The measurement model (summarized in Table 2) proposed factors that could affect change effort, and alternative ways the factors could be measured. Only measures that could be retrieved from version control systems and change trackers were considered, because this data source is usually available in well-organized software projects. Information from such tools may not perfectly capture the factors of interest; hence this data source introduces issues of construct validity. In some cases, we were able to use the qualitative data to mitigate such threats. For example, the interviews provided a subjective operationalization of *change experience* that allowed us to draw stronger conclusions about the effects of experience. There were also threats to construct validity in the qualitative coding schema. We attempted to mitigate this by reconsolidating the coding schema to reflect commonly used concepts within our field.

Requirement volatility would intuitively be considered to be high if extensive informal clarifications about change requirements were needed, even if the clarifications were documented by only a short summary in the change trackers. Therefore, our measures of requirement volatility, which relied on traces in such tools, may not perfectly capture this factor. The analysis of the interviews strengthened the proposal that requirement volatility was indeed an important factor, not always appropriately captured by the quantitative measures.

Code complexity cannot be fully captured by one or a few measures [65]. To judge, in a meaningful and repeatable manner, whether a piece of code is "more complex than" another piece of code, very specific criteria must be defined. Therefore, there were obvious construct validity threats in the measurement of complexity of *change sets* and *changed components*. The measures needed to be simple because they had to be compatible with the range of technologies that the projects used. Likewise, it is not obvious how measures of added, deleted and changed lines contribute to an aggregated measure of *change set size*, in particular when different technologies were involved, and it is not obvious which architectural unit to count when measuring *change set dispersion*.

Change experience was captured by counting the number of earlier check-ins to the version control systems. It is an obvious simplification that one check-in can be counted as one unit of experience. Moreover, when several developers were involved in a change, we used the average of the experience measures. This aggregation does not perfectly capture the concept of joint experience. It is possible that the relatively poor fit of the quantitative models was due to the inability to fully capture the intended factors by measures retrieved from version controls systems.

Internal validity. Internal validity of the quantitative results refers to the degree to which the variations to change effort in the investigated projects were caused by the proposed factors. Issues of internal validity are important when the context, tasks and procedures for allocating study units to groups cannot be controlled, which is the case with data that occurs naturally in software development projects. For example, a particularly skilled developer may require fewer requirement clarifications and less change effort than the average developer. In this case, the underlying effect is that of individual developers' skills than that of requirement volatility. Likewise, the data-driven analysis showed that changes that were local to one technology required less effort than other changes. However, it is possible that the underlying effect pertains to the specific technology, Java, that was used in most of the cases that involved only one technology. Qualitative data from developer interviews was useful to evaluate some of these threats. For example, the qualitative analysis suggested that a more fundamental factor than the effect of dispersion of changed code was the effect of dispersion of *intermediate* code that needed to be comprehended.

Another threat to internal validity was the possibility of shotgun correlations. In the data-driven analysis, a large number of factors and measures were tested. This increases the likelihood that one or more of the significant effects occurred due to chance, rather than to a true underlying effect. This risk was lower in the evidence-driven analysis, because this investigated the effect of a small set of factors and measures selected on the basis of existing empirical evidence.

A third type of threat to internal validity was the potential bias introduced by missing data points in the data set, see [66]. For project A, change effort was not recorded for around 10% of the actual changes that were performed. For project B, change effort was not recorded for 25% of the changes. Most of the missing data points were due to challenges with establishing the routines to track change effort and code changes. Because the data points that we did collect from the initial periods can be considered to be selected by random, we do not expect the missing data points to constitute a serious threat to internal validity.

The use of interviews introduced the possibility of researcher bias, consciously or unconsciously skewing the investigation to conform to the competencies, opinions, values or interests of the involved researchers. Although such threats apply to quantitative research as well, they can be particularly difficult to handle when subjectivity is involved. The developers may introduce conscious or unconscious biases in the qualitative data, for the same reasons as those mentioned above. Imperfect memory, lack of trust or other communication barriers between the interviewer and the interviewee may also introduce biases.

We believe that the strict focus on relatively small, cohesive tasks recently performed by the interviewee helped to mitigate such biases. To mitigate communication barriers, the interviewer made extensive efforts to be prepared for the interviews, and data from the version control systems and change trackers was readily available during the interviews to help the developers recollect details.

External validity. The ability to generalize results beyond the study context is one of the key concerns with case studies. Section 2.3 described the design elements introduced to interpret the results in a wider context. We believe that the lack of relevant theories on which to base the study proposals is a major obstacle to generalizing the results. In this situation, we chose to base the study proposals on a comprehensive review of earlier empirical studies with similar research questions. In this way, the study adds to the empirical foundation that eventually can provide more generally applicable evidence of how and when different factors affect change effort.

8 Conclusion, consequences and further work

Software engineering practices can be improved if they address factors that have been shown empirically to affect developers' effort during software evolution. In this study, we identified such factors by analyzing data about changes in two software organizations. Regression models were constructed to identify factors that correlated with change effort, and developer interviews were conducted to explore additional factors at play when the developers expended effort to perform change tasks. Central results were:

- The volatility of requirements had a large and consistent effect on effort in the quantitative models. The effect was particularly large when volatility was due to difficulties in anticipating side effects of a change. Such difficulties also resulted in errors by omission, which in turn were particularly expensive to correct.
- The dispersion of changed code also had a large and consistent effect on change effort in the quantitative models. The quantitative models indicated that dispersion of the directly affected code was important. The qualitative analysis indicated that the dispersion of *intermediate* code was a more fundamental factor that affected change effort, due to comprehension effort.
- The experience that developers had in changing the system or parts of the system seemed to have little effect in the quantitative models. However, the qualitative analysis showed that this factor was indeed important in individual cases.

Because these results are also consistent with results from earlier empirical studies, we suggest that these (admittedly quite course-grained) factors should be given consideration when attempting to improve software engineering practices.

The specific analyses of the two projects provided additional and more fine-grained results. In one project, changes that concerned only one technology required considerably less effort. The analysis of estimation accuracy indicated that this factor was not sufficiently accounted for when developers made their estimates. This exemplifies how projects can benefit from analyzing data from their version control systems and change trackers to improve their estimation practices.

One important direction for further work is to investigate further the causal relationships that are in play when developers perform change tasks. Interviewing developers about recent changes was an effective method for making tentative suggestions about such relationships. However, studies that control possibly confounding factors should be conducted before firm conclusions are drawn. It is also necessary to paint a richer picture of how context factors, such as size and type of the system, influence change effort. Ultimately, the empirical results could be aggregated into a *theory on software change effort*, which would define invariant knowledge about software evolution, and be immediately useful for practitioners within the field.

Acknowledgements We are indebted to the managers and developers at Esito and Know IT who provided us with high quality empirical data. We thank Audris Mockus for his comprehensive feedback on two early drafts of this paper. Dag Sjøberg and Magne Jørgensen provided strategic advice on the research approach. Jo Hannay and Stein Grimstad contributed with insightful comments in discussions about this work. Finally, we are grateful to the <removed at submission time> for funding the Ph.D. position of the first author of this paper.

Appendix A

Interview guide

Part 1. (Only in first interview with each developers - Information about the purpose of the research. Agree on procedures, confidentiality voluntariness, audio-recording).
Question: Can you describe your work and your role in the project?
Part 2. Project context (factors intrinsic to the time period covered by the changes under discussion)
How would you describe the project and your work in the last time period? Did any particular event require special focus in the period?
For each change (CR-nnnn, CR-nnnn, CR-nnnn….,)

Part 3. Measurement control (change effort and name of changed components shown to the interviewee)
Are change effort and code changes correctly registered?
Part 4. Change request characteristics (change tracker information shown on screen to support discussion)
Can you describe the change from the viewpoint of the user? Why was the change needed?
Part 5. General cost factors
Can you roughly indicate how the X hours were distributed on different activities?
Part 6. Properties of relevant code (output from *windiff* showed on screen to support the discussions)
Can you summarize the changes that you made to the components?
What can you say about the code that was relevant for the change? Was it easy or difficult to understand and make changes to the code?
Part 7. Stability
Did you go through several iterations before you reached the final solution? If so, why?
Did anything not go as expected?
How did you proceed to test the change?
Go to Part 3 for next change

Part 8. Concluding remarks
Do you think this interview covered your activities during the last period?

Appendix B

Effort drivers and effort savers for individual changes

Table B1 Underestimated changes, project A

CR#	Pred.	*Change description*	Code
	Actual	Developer statement (translated and condensed)	
A4155	9.4	*Ensure consistency between reported grants, expenses and accounts*	C3
	19	Input control spans fields were spread over three tables	D2
		Input control should be conditioned by check-button state	D2
		One file was not tagged	n.a.
		Unintended consequence - too strict input control	R1
A4666	10.2	*Calculate and show deviation between grants and expenses*	C3
	21.5	Remove validation. Conditioned by check-button state	D2
		The original specification was not very detailed	R1
		Many Javascript changes. Have not very much Javascript experience	T0
		Less debugging support in XSL/Javascript, use write-statements	T3
A4569	2.3	*Check special case when saving user access rights*	C1
	5	Discussion about on whether and how it should be done	R1
A4568	1.6	*Fix programming error, used wrong variable*	C2
	3.5	Code reading not successful, needed to execute/debug	U1
		No direct debug support	T3
A4557	4.5	*Change trigger rule for starting timer for reminder*	C1
	10	Workflow tool: Could not use out-of-the box support, special code needed	T1
		Defect discovered in tool, needed to create a work-around	T2
		Difficult to test when actions are based on time triggers, must manipulate database	V1
A4427	18.0	*Create web based view into research application data*	C4
	41	Choose technical solution, chose xslt	D2
		Mechanism for access to details	D2
		Many rounds of feedback on page layout details	R1
A4282	9.0	*Serverside input validation of application*	C3
	21	Difficult to comprehend external framework, recursive functions, difficult to follow control flow	U1
		XPath not well known	T0
A518	2.0	*Correction of correction*	C1
	5	Complex state due to collaborating screens makes the code difficult to understand	U1
A512	4.0	*Handle unexpected user input, empty fields*	C1
	10.5	Had to change the interface, create a new method, and send in another object	D2
		Needed to discuss which rules to implement	R1
A4211	6.9	*Transfer data to external system, Check social security number*	C3
	19	Re-implement algorithm in Javascript	D3
A4438	1.6	*Error in pageflow on validation error*	C1
	5	Internal state needed to be set correctly, needed time to realize this	U1
		Separate debugging tool, could not use eclipse	T3
A4461	33	*Get new mechanisms for persistence in place, integrate design* and runtime tools	C4
	117	Defects in Genova	T2
		Unfamilar with Hibernate	T0
		Impossible to debug in Eclipse	T3
		Time consuming to deploy for debug in other tool	T3
		Unstable debugging tool	T2
A4122	7.0	*Assumptions of max 10 years broken in GUI*	C4
	20.5	Upfront effort on analysis/design	R1/D2
		Implement scrolling mechanism in GUI	D2

Table B2 Underestimated changes, project B

CR#	Pred.	*Change description*	Code
	Actual	Developer statement (translated and condensed)	
B4189	3	*Defect, wrong assumption that object was already created*	C1
	6	It was not well specified	R1
		Use log and debug to reproduce the state that gives the error	U2
B3777	23	*Special key for Oslo-ticket*	C4
	48	Requirement clarifications needed	R1
		Iterative design	D2
B4062	2.8	*Fix state action: Must ensure correct printout, dependent on the type of area* Difficult to	C1
	6	determine full state	U1
		Difficult algorithm	D2
B4188	3.4	*Defect: Must check for events on ticket before setting to unused after cancelling*	C1
	7.5	Difficult to determine when to perform which action	U2

		Testing dependent on time	V1
B3935	4.2 9.5	*Write receipt on sale from MT, from popup* Unfamiliar with print code Need to restructure	C4 U3 D2
B4260	3.4 8	Logging of two events needed to separated, because they did not always happen together	C1 U1 D2
B4089	3.6 9.5	*Need to cancel sale on e-ticket on technical cancelling* Difficult to identify the part of the code that handled technical cancelling, due to naming Unfamiliar with the code	C1 U2 U3
B4157	12.4 32.5	*Special key for prefer2travel* Difficult because it was not very well specified Difficult to know which part of the code to change Difficult to comprehend what is part of the create sale transaction Earlier attempt to start coding	C4 R1 U3 U1 D2
B3278	12.4 20	*Corrected defect due to string-number conversion* Assumed to be correct, unstable api-call	C1 T2

Table B3 Overestimated changes, project A

CR#	Pred. Actual	*Change description* Developer statement (translated and condensed)	Code
A4555	6.4 2.5	*Wrong text substitution in emails, text retrieved from content server* Easily recreated	C2 U2
A4531	2.8 1	*Changes to CSS and layout*	C3 D1
A4434	2.9 1	*Set input field type in xml's*	C3 D1
A4426	1.6 0.5	*Change name of GUI--field*	C3 D1
A4607	1.6 0.5	*Missed specific state and action* Simple check on condition Needed only one place	C1 U2 D3
A4539	1.6 0.5	*Wrong action, should not update last change by on automatic change* Easy to identify	C1 U2
A4578	1.7 0.5	*Moved comparator method between class* One user only	C1 D3
A4330	7 2	Display differently dependent on research application status Change to property-files	C4 D1
A4279	7.6 2	New CSS definition for read-only fields	C3 D1 D3
A4369	4.3 1	*Add one element to transfer to external system* Have worked with this program before Had a framework available for testing Knew the class where the change needed to be done	C1 U3 V1 D3
A4596	2.3 5	*Name change on label* I knew very well how to make the change	C3 D1 U3
A4500	2.3 0.5	*Keep relations to organization when creating revised application* It was a small and local change Reused a method that could do this	C4 D3 D2
A4542	7.6 1	*Changes to fonts, small error correction* We needed to create a new CSS class	C2 D1 D3
A4559	4.1 0.5	*Error in text*	C2 D1 D3
A4547	1.7 0.17	*New button in webscreen* Only needed to add a button	C4 D1,T 1,D3
A4414	2.4 0.17	*Sort column in table* Framework contained the exact needed API	C3 T1
A4584	111 39	*Request for approval that application could be visible to others* It was very well specified Spanned many classes	C4 R1 P6

Table B4 Overestimated changes, project B

CR#	Pred. Actual	*Change description* Developer statement (translated and condensed)	Code
B4367	3.3 1	*Remove logging calls* Many similar changes	C3 U1
B4366	3	Not good enough data. Interviewee was instructed what to do	n.a.

	1		
B3765	17 6	*Move and split location of data attribute* Well specified	C3 R1
B3928	2.4 1	*Reset screen on error* Make call to predefined function on error	C3 D2
B4022	6,1 2.5	Check that more than 2000 points are not sold Local change	C4 P6
B4233	2.8 0.5	Logging level adjusted Simple, local change	C1 P6

References

1. Lehman MM, Ramil JF, Wernick PD, Perry DE, and Turski WM. Metrics and laws of software evolution - the nineties view. *Proceedings of the 4th International Symposium on Software Metrics.* IEEE Computer Society Press: Los Alamitos CA, 1997; 20-32.
2. Banker RD, Datar SM, Kemerer CF, and Zweig D. Software complexity and maintenance costs. *Communications of the ACM* 1993; **36**(11):81-94.
3. Bhatt P, Shroff G, Anantaram C, and Misra AK. An influence model for factors in outsourced software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice* 2006; **18**(6):385-423.
4. Krishnan MS, Kriebel CH, Kekre S, and Mukhopadhyay. T. An empirical analysis of productivity and quality in software products. *Management Science* 2000; **46**(6):745-759.
5. Lientz BP. Issues in software maintenance. *ACM Computing Surveys* 1983; **15**(3):271-278.
6. Hayes JH, Patel SC, and Zhao L. A metrics-based software maintenance effort model. *Proceedings of the 8th European Conference on Software Maintenance and Reengineering.* IEEE Computer Society Press: Los Alamitos CA, 2004; 254-258.
7. Kemerer C. Software complexity and software maintenance: A survey of empirical research. *Annals of Software Engineering* 1995; **1**(1):1-22.
8. Munson JC and Elbaum SG. Code churn: A measure for estimating the impact of code change. *Proceedings of the 14th International Conference on Software Maintenance.* IEEE Computer Society Press: Los Alamitos CA, 1998; 24-31.
9. Détienne F and Bott F, *Software design - cognitive aspects*. London: Springer-Verlag, 2002.
10. Benestad HC, Anda BC, and Arisholm E, "A systematic review of empirical software engineering studies that analyze individual changes," Simula Research Laboratory Technical Report 2008-05, 2008.
11. Yin RK. Designing case studies in *Case study research: Design and methods.* Sage Publications:Thousand Oaks, CA, 2003; 19-53.
12. Benestad HC, Arisholm E, and Sjøberg D. How to recruit professionals as subjects in software engineering experiments. *Information Systems Research in Scandinavia (IRIS), Kristiansand, Norway* 2005;
13. https://www.forskningsradet.no/mittNettstedWeb/common/security/login.jsp?setLocale=en
14. Norwegian State Railways, http://www.nsb.no/about_nsb/
15. IBM Rational ClearCase, http://www-306.ibm.com/software/awdtools/clearcase/cclt/
16. CVS, http://www.nongnu.org/cvs/
17. http://www.atlassian.com/software/jira/
18. Schneidewind NF. Investigation of the risk to software reliability and maintainability of requirements changes. *Proceedings of the 2001 International Conference on Software Maintenance.* IEEE Computer Society Press: Los Alamitos CA, 2001; 127-136.
19. Niessink F and van Vliet H. Two case studies in measuring software maintenance effort. *Proceedings of the 14th International Conference on Software Maintenance.* IEEE Computer Society Press: Los Alamitos CA, 1998; 76–85.
20. Evanco WM. Analyzing change effort in software during development. *Proceedings of the 6th International Symposium on Software Metrics (METRICS99).*1999; 179-188.
21. Graves TL and Mockus A. Inferring change effort from configuration management databases. *Proceedings of the 5th International Symposium on Software Metrics.* IEEE Computer Society Press: Los Alamitos CA, 1998; 267–273.
22. Jørgensen M. Experience with the accuracy of software maintenance task effort prediction models. *IEEE Transactions on Software Engineering* 1995; **21**(8):674-681.
23. Arisholm E. Empirical assessment of the impact of structural properties on the changeability of object-oriented software. *Information and Software Technology* 2006; **48**(11):1046-1055.
24. Atkins DL, Ball T, Graves TL, and Mockus A. Using version control data to evaluate the impact of software tools: A case study of the version editor. *IEEE Transactions on Software Engineering* 2002; **28**(7):625-637.
25. Briand LC and Basili VR. A classification procedure for the effective management of changes during the maintenance process. *Proceedings of the 1992 Conference on Software Maintenance.* IEEE Computer Society Press: Los Alamitos CA, 1992; 328-336.
26. Hunt JW and McIlroy MD, "An algorithm for differential file comparison," in *Computing Science Technical Report 41, Bell Laboratories*, 1975.
27. Moløkken-Østvold K, Haugen NC, and Benestad HC. Using planning poker for combining expert estimates in software projects. *Accepted for publication in Journal of Systems and Software* 2008;
28. Fluri B and Gall HC. Classifying change types for qualifying change couplings. *Proceedings of the 14th International Conference on Program Comprehension (ICPC).*2006; 35-45.
29. Polo M, Piattini M, and Ruiz F. Using code metrics to predict maintenance of legacy programs: A case study. *Proceedings of the 2001 International Conference on Software Maintenance.* IEEE Computer Society Press: Los Alamitos CA, 2001; 202-208.
30. Swanson EB. The dimensions of maintenance. *Proceedings of the 2nd International Conference on Software Engineering.* IEEE Computer Society Press: Los Alamitos CA, 1976; 492-497.
31. Jørgensen M. An empirical study of software maintenance tasks. *Journal of Software Maintenance: Research and Practice* 1995; **7**(1):27-48.

32. Purushothaman R and Perry DE. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering* 2005; **31**(6):511-526.
33. Etzkorn L, Bansiya J, and Davis C. Design and code complexity metrics for oo classes. *Journal of Object-Oriented Programming* 1999; **12**(1):35-40.
34. Rajaraman C and Lyu MR. Reliability and maintainability related software coupling metrics in c++ programs. *Proceedings of the Third International Symposium on Software Reliability Engineering.*1992; 303-311.
35. Niessink F and van Vliet H. Predicting maintenance effort with function points. *Proceedings of the 1997 International Conference on Software Maintenance.* IEEE Computer Society Press: Los Alamitos CA, 1997; 32-39.
36. Eick SG, Graves TL, Karr AF, Marron JS, and Mockus A. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering* 2001; **27**(1):1-12.
37. Graves TL, Karr AF, Marron JS, and Siy H. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering* 2000; **26**(7):653-661.
38. DeMarco T and Lister T. Programmer performance and the effects of the workplace. *Proceedings of the Proceedings of the 8th international conference on Software engineering.*1985; 268-272.
39. Sackman H, Erikson WJ, and Grant EE. Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM* 1968; **11**(1):3-11.
40. Mockus A and Weiss DM. Predicting risk of software changes. *Bell Labs Technical Journal* 2000; **5**(2):169-180.
41. Myers RH, Montgomery DC, and Vining GG. The generalized linear model in *Generalized linear models with applications in engineering and the sciences.* Wiley Series in Probability and Statistics, 2001; 4-6.
42. http://www.transana.org/
43. Conte SD, Dunsmore HE, and Shen VY, *Software engineering metrics and models*: Benjamin-Cummings Publishing Co., Inc. Redwood City, CA, USA, 1986.
44. Jolliffe IT, *Principal component analysis*, 2nd ed. New York: Springer-Verlag, 2002.
45. Briand LC and Wüst J. The impact of design properties on development cost in object-oriented systems. *IEEE Transactions on Software Engineering* 2001; **27**(11):963-986.
46. von Mayrhauser A and Vans AM. Program comprehension during software maintenance and evolution. *Computer* 1995; **28**(8):44-55.
47. Koenemann J and Robertson SP. Expert problem solving strategies for program comprehension. *Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology.*1991; 125-130.
48. Dzidek WJ, Arisholm E, and Briand LC. A realistic empirical evaluation of the costs and benefits of uml in software maintenance. *IEEE Transactions on Software Engineering* 2008; **34**(3):407-432.
49. Arisholm E, Briand LC, and Føyen A. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering* 2004; **30**(8):491-506.
50. Détienne F and Bott F. Influence of the task in *Software design - cognitive aspects.* Springer-Verlag, 2002; 105-110.
51. Gamma E, Helm R, Johnson R, and Vlissides J, *Design patterns: Elements of reusable object-oriented software*: Addison-Wesley, 1995.
52. Herbsleb JD and Mockus A. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering* 2003; **29**(6):481-494.
53. Wirfs-Brock R and McKean A, *Object design: Roles, responsibilities, and collaborations*: Addison-Wesley Professional, 2003.
54. Détienne F and Bott F. Discontinuities and delocalized plans in *Software design - cognitive aspects.* Springer-Verlag, 2002; 113-114.
55. Arisholm E and Sjøberg DIK. Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Transactions on Software Engineering* 2004; **30**(8):521-534.
56. Kern J and Garret C: Effective Sequence Diagram Generation, 2003.
57. Briand LC, Labiche Y, and Miao Y. Towards the reverse engineering of uml sequence diagrams. *Proceedings of the 10th Working Conference on Reverse Engineering, WCRE 2003.*2003; 57-66.
58. Cohn M, *Agile estimating and planning*: Pearson Education, Inc. Boston, MA, 2006.
59. Kitchenham BA, Dybå T, and Jørgensen M. Evidence-based software engineering. *Proceedings of the 26th International Conference on Software Engineering (ICSE).* IEEE Computer Society, 2004; 273-281.
60. Kitchenham BA, Pleeger SL, Pickard LM, Jones PW, Hoaglin DC, El Emam K, and Rosenberg J. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering* 2002; **12**(4):1106-1125.
61. Seaman CB. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering* 1999; **25**(4):557-572.
62. Perry DE, Porter AA, and Votta LG. Empirical studies of software engineering: A roadmap. *Proceedings of the Conference on The Future of Software Engineering.*2000; 345-355.
63. Birk A, Dingsøyr T, and Stålhane T. Postmortem: Never leave a project without it. *IEEE Software* 2002; **19**(3):43-45.
64. Derby E and Larsen D, *Agile retrospectives: Making good teams great*: Raleigh, NC: Pragmatic Bookshelf, 2006.
65. Fenton N. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering* 1994; **20**(3):199-205.
66. Mockus A. Missing data in software engineering in *Guide to advanced empirical software engineering.* 2000; 185-200.