Information and Software Technology 52 (2010) 110–125

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Change profiles of a reused class framework vs. two of its applications

Anita Gupta [a,*], Jingyue Li [a], Reidar Conradi [a], Harald Rønneberg [b], Einar Landre [b]

[a] Department of Computer and Information Science (IDI), Norwegian University of Science and Technology (NTNU), Trondheim, Norway
[b] StatoilHydro ASA, KTJ/IT, Forus, Stavanger, Norway

ARTICLE INFO

Article history:
Received 1 July 2008
Received in revised form 31 July 2009
Accepted 8 August 2009
Available online 14 August 2009

Keywords:
Software reuse
Software change profile
Case study
Class framework

ABSTRACT

Software reuse is expected to improve software productivity and quality. Although many empirical studies have investigated the benefits and challenges of software reuse from development viewpoints, few studies have explored reuse from the perspective of maintenance. This paper reports on a case study that compares software changes during the maintenance and evolution phases of a reused Java class framework with two applications that are reusing the framework. The results reveal that: (1) The reused framework is more stable, in terms of change density, than the two applications that are reusing it. (2) The reused framework has profiles for change types that are similar to those of the applications, where perfective changes dominate. (3) The maintenance and evolution lifecycle of both the reused framework and its applications is the same: initial development, followed by a stage with extending capabilities and functionality to meet user needs, then a stage in which only minor defect repairs are made, and finally, phase-out. However, the reused framework goes faster from the stage of extending capabilities to the stage in which only minor defect repairs are made than its applications. (4) We have validated that several factors, such as are functionalities, development practice, complexity, size, and age, have affected the change densities and change profiles of the framework and applications. Thus, all these factors must be considered to predict change profiles in the maintenance and evolution phase of software.

1. Introduction

After a software system is delivered to the client for use, it will evolve and be maintained continuously until it is replaced or discarded [4]. Pigoski [32] found that the percentage of the IT industry's expenditure on maintenance was 40% in the early 1970s, 55% in the early 1980s, and 90% in the early 1990s. Krogstie et al. [22] conducted a survey that investigated the development and maintenance of business software in Norway. The same survey was performed in 1993 and 1998. The results show that overall, about 40% of available time is spent on maintenance. Software reuse is expected to utilize past accomplishments, to facilitate software development productivity, and to improve the quality of the developed software system [26]. Several studies have investigated whether software reuse and component-based development have the potential to facilitate the maintenance and evolution of a system. They concluded that reused components are more stable than non-reusable ones [13,2,31,40,49]. However, one study concludes the opposite, stating that reused components with major revisions later will need more changes per source line than newly developed components [40].

To investigate the benefits and challenges of software reuse with respect to maintenance, we performed an industrial case study to compare the actual changes performed on a reusable framework called the Java Enterprise Framework (JEF). This framework is developed by the IT-department of a large Norwegian Oil and Gas company. It is reused "as-is" by two applications, digital cargo files (DCF) and shipment and allocation (S&A) in the same company. We formulated two research questions:

RQ1: Whether the reused software experiences fewer or more changes than its applications and the likely reasons for the differences or similarities.
RQ2: Whether the reused software experiences the same profile of changes over time with the software reusing it and the reasons for the differences and similarities.

We randomly selected files from the three systems, classified all changes on these files after the first release, and compared the distribution of different types of changes (perfective, corrective, adaptive, and preventive) in general and over time. In addition, we performed a root cause analysis (RCA) to interpret the results. The novelties of the study are

* Corresponding author.
E-mail addresses: anitaash@idi.ntnu.no (A. Gupta), jingyue@idi.ntnu.no (J. Li), conradi@idi.ntnu.no (R. Conradi), haro@statoilhydro.com (H. Rønneberg), einla@statoilhydro.com (E. Landre).

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2009.08.002

- It compared the maintenance activities of reused software vs. non-reused software by classifying changes into perfective, corrective, adaptive, and preventive types of change.
- It compared the change profile of reused and non-reused software over time.
- It used root cause analysis to investigate the reasons for software evolution and maintenance activities.

This study contributes to the understanding of software reuse and software changes, and concludes that:

- The reused framework has a lower change density than the two applications reusing it.
- Perfective changes (i.e. caused by new or changed requirements, as well as optimizations) constitute the highest percentage of changes in both the reused framework and in the applications reusing it.
- Both the reused framework and the applications experienced the following lifecycle: initial development, then extending the capabilities and functionalities of the system to meet user needs, followed by repairing minor defects. However, the reused framework experienced only one such lifecycle, while the applications experienced several.
- Kemerer and Slaughter [20] concludes that five main factors (i.e. software functionality, software complexity, development practices, software size, and software age) will affect possible maintenance activities. Regarding the change densities and change profiles of both the framework and the applications, those of Kemerer's factors [20] that affect the maintenance activity most in our case are functionality and development practices, followed closely by complexity. The factors that affect it least in our case are age and size. The functionalities and development practices of the software usually influence the future change density and the type of change (perfective, corrective, adaptive, and preventive).

The remainder of the paper is structured as follows. Section 2 presents related work. Section 3 presents the motivation for the research and research questions. Section 4 describes the research design. Section 5 presents the data analysis results. Section 6 discusses these results. Section 7 concludes.

2. Related work

Understanding the issues related to changes involved in the evolution and maintenance of software has been a focus of software engineering study since the 1970s. The goal has been to identify the origin of changes, as well as their type, relative frequency of occurrence, and effort required to make them. Software changes are important because they account for a major part of software costs. At the same time, they are necessary. The ability to alter software quickly and reliably means that businesses can take advantage of new opportunities and can remain competitive [4]. Lehman [24] carried out the first empirical studies of software changes, finding that systems that operate in the real world have to be adapted continuously. Otherwise, their usability and relevance decreases rapidly. Software systems usually need to be changed during their lifetime because the original requirements may change to reflect changing business, user, and client needs [33].

2.1. Studies on the distributions of different types of software changes

One kind of study that has been performed on software changes examines the *static* aspect of changes, i.e. the distribution of different kinds of change, or the distribution of effort spent on performing different kinds of change. Table 1 summarizes distribution and definition of different type of changes in these studies. The design and other observations of these studies are presented in Appendix A.

A close investigation of studies in Table 1 reveals that:

- Different studies classify changes differently, as noticed by [8].
  - Four studies classified changes into four categories: adaptive, corrective, perfective, and preventive [18,36,31,23].
  - Several studies did not include preventive changes and classified the changes into adaptive, corrective, and perfective, with a fourth category of user support in [1], inspection in [29], and "other" in [25,47,3,38].
  - One study classified changes into planned enhancement, requirement modifications, optimization, and "other" [11].
  - One study classified changes into user support, repair, and enhancement [5].

- Definitions of different types of change are slightly different. For example, perfective change is defined as user enhancements, improved documentation, and recoding for computational efficiency in [25], and as restructuring the code to accommodate future changes in [29]. It is also defined as encompassing new or changed requirements (expanded system requirements) as well as optimization in [43,31]. In [47], it is defined as enhancements, tuning, and reengineering.
- The distributions of different types of change are not the same for different systems. Studies [25,47,3] found that perfective changes were the most frequent. However, perfective changes in the system in [29] were the least frequent. Studies [5,29] concluded that corrective changes were the most frequent ones. Other studies, such as [1,43], found that adaptive changes were the most frequent changes.

2.2. Studies on software changes over time

Another kind of study on software changes investigated how the changes vary over time (the *longitudinal* aspect.) Gefen and Schneberger [14] examined an information system for 29 months and reported that in the first stage, the software was stabilized within the framework of its original specifications and changes were centered on corrective modifications. In the second period, the software was improved and new functions were added to the original framework. In the third period, the system was expanded beyond its original specifications by adding many new applications. Burch and Kung [5] studied a large application for 67 months and reported that user support reaches its peak in the first stage. Repair is prevalent in the second stage and enhancement is the dominant factor in the third stage. Rajlich and Bennett [4] proposed a stage model to describe the lifecycle of a software system, as shown in Fig. 1. According to that model, the software life cycle consists of five distinct stages:

- *Initial development.* Engineers develop the system's first functioning version.
- *Evolution.* Engineers extend the capabilities and functionality of the system to meet user needs, possibly in major ways.
- *Servicing.* Engineers repair minor defects and make simple functional changes.
- *Phase-out.* The company decides not to undertake any more servicing, seeking to generate revenues from the existing system for as long as possible.
- *Close-down.* The company withdraws the system.

A variation of the stage model is the versioned staged model [4], also shown in Fig. 1. The backbone of the versioned staged model is

Table 1
Studies related to distributions of different types of changes.

Studies	Definition and distribution of different types of changes
Lientz et al. [25]	– 60% *perfective* (enhancements and speed performance) – 18% *adaptive* (changes to data inputs and files) – 17% *corrective* (emergency fixes and debugging) – 4% *other* (no description given)
Abran and Nguyenkim [1]	– 60% *adaptive* (changes to data inputs and files) – 21% *corrective* (emergency fixes and debugging) – 3% *perfective* (enhancements and speed performance) – 15% *user support* (handle user requests of application rules and behavior, requests for work estimates, requests for preliminary analysis)
Yip and Lam [47]	– *perfective* (40% enhancements, 7% tuning, and 6% reengineering) – 16% *corrective* (correct faults) – 10% *adaptive* (adaptation to new environment) – *other* (13% answering questions and 9% documentation)
Jørgensen [18]	*Results of interviews with managers:* – 44% *perfective* (changes in user requirements) – 29% *adaptive* (make software usable in a changed environment) – 19% *corrective* (correct faults) – 8% *preventive* (preventing problems before they occur) *Results of interviews with maintainers:* – 45% *perfective* – 40% *adaptive* – 9% *corrective* – 6% *preventive*
Basili [3]	– 61% *perfective* (improve system attributes and add new functionality) – 20% *other* (e.g. management, meeting, etc.) – 14% *corrective* (correct faults) – 5% *adaptive* (adapt system to new environment)
Burch and Kung [5]	– 49% *repair* (fixing bugs) – 26% *enhancement* (add or modify functionalities) – 25% *user support* (consulting and answering user requests)
Sousa and Moreira [43]	– 49% *adaptive* (changes in platform) – 36% *corrective* (error modifications) – 14% *perfective* (expand system requirements and optimization) – 2% *preventive* (future maintenance action)
Evanco [11]	– 31% *planned enhancements* (anticipated at the start of development) – 30% *other* (code debugging, enhancements and maintainability) – 29% *requirements modifications*(implementation of requirement changes) – 10% *optimization* (optimize software performance)
Mockus and Votta [29]	– 46% *corrective* (fixing faults) – 45% *adaptive* (adding new features) – 5% *inspection* (code checking to figure errors) – 4% *perfective* (code restructuring)
Satpathy et al. [36]	– 38% *perfective* (optimization, restructuring and adding new functionalities) – 31% *adaptive* (adapting to changed environments) – 23% *preventive*(preventing malfunctions and improving maintainability) – 8% *corrective*(correcting problems)
Schach et al. [38]	The analysis and collection of data were performed at two levels, using the same definition as Lientz et al. [25]: (1) *change log level*, that is, each entry in the change log was regarded as one unit of change. (2) *module level*, that is, all the changes made to a module were regarded as a single unit of maintenance. *Change log level:* – 57% *corrective* – 39% *perfective* – 2.4% *other* – 2.2% *adaptive* *Code module level:* – 53% *corrective* – 36% *perfective* – 4% *adaptive* – 0% *other*
Mohagheghi and Conradi [31]	– 61% *perfective* (new or changed requirements as well as optimization) – 19% *adaptive* (adapting to new platforms or environments) – 16% *preventive* (restructuring and reengineering) – 4% *other* (saving money/effort) Corrective changes are reported elsewhere
Lee and Jefferson [23]	*Based on Swanson's [49] definitions:* – 62% *perfective* – 32% *corrective* – 6% *adaptive* *Based on Kitchenham's [21] ontology:* – 68% *enhanced maintenance* – 32% *corrective*

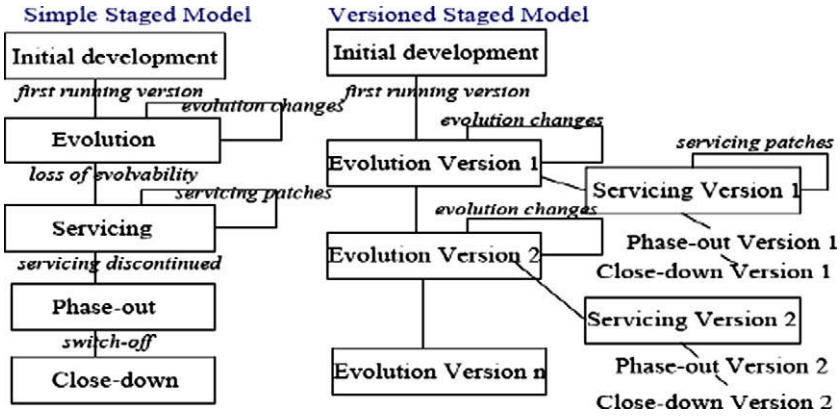

Fig. 1. Simple staged model and versioned staged model [4].

Table 2
Studies comparing changes of reusable components vs. those in non-reusable ones.

Quality focus	Quality measures	Conclusion
Change density	Number of change requests per source code line	Reused components are more stable in terms of volume of code modified between releases [30]
	Percentage of code-line changes (enhancement or repair)	The modules reused with major revision (> = 25% revision) had the most code changes per SLOC [40]
Number of changes	The number of changes (enhancement or repair) to a module	More reuse results in fewer changes [13]
Amount of modified code	Size of modified or new/deleted code/total size of code per component between releases	Non-reused components are modified more than reused ones [31]
Number of change scenarios	Number of changes to which a software system is exposed (e.g. adding communication protocols, porting to new platforms, issues related to the database manager, etc.)	Reusing components and a framework resulted in increased maintainability in terms of cost of implementing change scenarios [2]

the evolution stage. At certain intervals, a company completes a version of its software and releases it to clients. Evolution continues, with the company eventually releasing another version and only servicing the previous version.

2.3. Studies on software changes and software reuse

A few studies [13,2,30,40] have examined the possible influence of software reuse on the changes of a system, as shown in Table 2.

Although most studies [13,2,30] in Table 2 conclude that *software reuse is significantly correlated to fewer changes*, one study observed that a reused module that undergoes major revision has the most changes per source line [40]. A close investigation of the studies in Table 2 further illustrates that:

– None of the studies performed detailed analyses (as was the case with studies listed in Table 1) of changes. "Detailed analysis" here refers to dividing the changes into different types and comparing the distribution of the changes according to type. Several factors, e.g. complexity, functionality, development practice, age, and size may determine the profile of software maintenance [20]. Thus, *comparing only the number or density of the defects is not sufficient to warrant the conclusion that software reuse is significantly correlated to fewer changes*. Thus, further study is needed to investigate the relation between software reuse and software changes of different types.

3. Research motivation and research questions

No study listed in Table 2 performed analysis similar to that in [14,5], i.e. compared the changes of reused software and the software reusing it over time. Thus, it remains an open question whether reused software and its applications follow similar or different change profiles or software lifecycles [4] over time. Answers to this question would make it easier for software maintainers to plan maintenance effort according to possible change profiles of reused software and software reusing it.

To determine whether *software reuse actually leads to fewer software changes*, we decided to compare the software changes that are made to reusable vs. non-reusable software. We investigated two research questions below to address the limitations of the studies in Table 2.

3.1. Research questions

Research question *RQ1* is: *Whether the reused software experiences fewer or more changes than its applications, and the likely reasons for the differences or similarities.* To examine this research question in details, we designed three sub-questions as follows:

RQ1.1. What are the total change densities of reused software and of non-reused software?
RQ1.2. What are the percentages covered by individual change types of reused software and of non-reused software?
RQ1.3. What are the reasons for the answers to research questions *RQ1.1* and *RQ1.2*?

The research questions *RQ2* is formulated as: *Whether the reused software experiences the same profile of changes over time with the software reusing it, and the reasons for the differences and similarities.* To examine this research question in detail, we designed two sub-questions as follows:

RQ2.1. What are possible differences of change profile of software being reused and software reusing it?
RQ2.2. What are the reasons for the answers to *RQ2.1*?

4. Research design

To answer our research questions, we investigated three software systems from the largest Norwegian oil and gas company, StatoilHydro ASA. In this section, we first introduce the company, the three systems, and software change data of these systems. After that, we describe how the change data were analyzed and how the follow-up RCAs [6] were performed.

4.1. Data collection

4.1.1. The investigated company

StatoilHydro ASA is a Norwegian Oil and Gas company. The company has its headquarters in Norway and has branches in 40 countries. It has more than 30,000 employees in total. The IT-department of the company is responsible for developing and delivering domain-specific software to the host organization, so that key business areas of Oil and Gas can become more flexible and efficient in their standard operations. It is also responsible for the operation and support of IT systems. This department consists of approximately 100 developers, who are located mainly in Norway. In addition, the company subcontracts many software development and operations to consulting (software) companies. These subcontracting operations may involve over 1000 IT specialists.

4.1.2. The software systems investigated

The company initiated its reuse strategy in 2003 with pre-studies. Then, a reusable software framework was developed. This framework (Java class library) is based on J2EE (Java 2 Enterprise Edition), and is a Java class framework for developing enterprise applications. Thus, the framework is called the "JEF framework" (hereafter, JEF). The JEF consists of seven separate components or modules (i.e. each component or modules consist of various library classes) that can be reused separately or together.

JEF release 1 was finished around June 2005. Physical Deal Maintenance (PDM) was the first application to use it in summer of 2005. In this period, some weaknesses in JEF were discovered. Changes in response to these weaknesses were incorporated into JEF release 2 in September 2005. The DCF application reused JEF release 2 during late summer and autumn of 2005. After DCF reused the JEF, further minor changes were made to the framework. These changes were finished by early November 2005, when the third JEF release was deployed. The second application, S&A, was developed during early 2006 and reused JEF release 3.

The relations between the JEF and applications using/reusing it are shown in Fig. 2. Detailed information of JEF, DCF and S&A are presented in Appendix B. The company has the same test team and the same test coverage for both reusable and non-reusable software. For instance, for unit testing, 85% of the code lines are executed by unit tests to make sure the code works as expected. AS PDM was the first application to *use* JEF. It did not *reuse* it (like DCF and S&A). Hence, changes from PDM were not analyzed in this study.

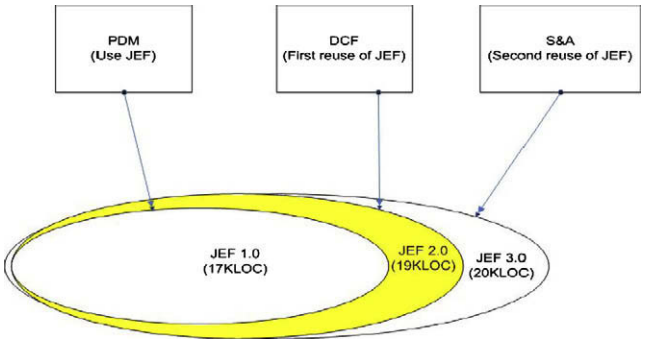

Fig. 2. The relation between JEF, DCF, and S&A.

4.1.3. Collected software change data

To handle changes in requirements or implemented artifacts, change requests (CRs) are written (by test manager or developers) and stored in the Rational ClearQuest tool. Examples of change requests are to add, modify or delete functionality; solve a problem with major design impact; or adapt to changes from, for example, JEF component interfaces. The project leader in StatoilHydro ASA constitutes the Change Control Board (CCB). The CCB is responsible for approving or rejecting a CR, and distributing the approved change requests (from Rational ClearQuest) among the developers. After the approved CRs have been distributed, the developers access the source files in the version control system, i.e. Rational ClearCase, to make the necessary changes. When implementing the changes, the developers follow the following steps:

- They check-out the files that correspond to the CR that they are working on.
- They implement changes on the checked-out files, possibly locking the branch that they are working on.
- They give the file a change description, which is a thorough description that elaborates on what changes they have made, and a time and date (timestamp).
- Finally, they check the changed files back into Rational ClearCase.

Due to the fact that Rational ClearCase captures information about all source code and other software changes, the first and second authors of the paper manually extracted the following data for each file (of JEF, DCF, and S&A) from Rational ClearCase: an ID, a filename with all its version numbers and the corresponding check-in timestamp (includes only the date), change description (prose description of the change), and location of the changes (i.e. the name of the components on which the changes were implemented).

We finished the data collection from three systems one by one. We finished data collection at 9th January 2007 for JEF, DCF at 20th September, 2007, and S&A at 6th February 2008. The release dates of the three systems and the last date of data collection from these systems are shown in Table 3. At the last date of data collection,

Table 3
The release date of the three systems and the last date of data collection.

System	Release 1	Release 2	Release 3	The last date of data collection
JEF	14 June 2005	9 September 2005	8 November 2005	9 January 2007
DCF	1 August 2005	14 November 2005	8 May 2006	20 September 2007
S&A	2 May 2006	6 February 2007	12 December 2007	6 February 2008

there were in total (excluding *test.java for unit tests) 634 java source files for the JEF, 694 for DCF, and 2823 for S&A.

To collect representative samples from the three systems for analysis, we first selected all components in each system, and then randomly selected a subset of the files from each component. When we collected files, we decided to have a threshold value of 10 files. If a component had less than 10 files, we included all the.-java files. If there were more than 10 ·java files in a certain component, we picked a random subset of files from it. For JEF and DCF, a "sampling calculator" [27] was used to calculate a sufficient sample size (confidence level was 95% and confidence interval was 3%). As an example of our procedure, the component JEFClient had 195 files. Using the calculated sample size from the sampling calculator, which was 165 files, we then randomly selected 165 files from the JEFClient to include in the dataset. For S&A, we did not use the sampling calculator to calculate the sample size, because it ended up with too many files to beyond our data collection capabilities. Instead, we decided to choose 25% of the files from S&A. At the end, we collected data from 442 files for the JEF, 501 for DCF, and 700 for S&A.

4.2. Data analysis

Although most previous studies shown in Table 2 on software changes are using non-commented source-line-of-code (NSLOC) as a measure for the change density, we decided to use more object-oriented (OO) metrics, as all our three systems are using Java as the programming language. Commonly used OO metrics, such as weighted methods per class (WMC), coupling between objects (CBO), and depth of inheritance tree (DIT), all proposed by Chidamber and Kemerer [9] have been used to measure the complexity of the system or class. However, those metrics have been mainly used to predict defect-prone components [48] [19,50]. Few of these metrics have been validated to explore reuse potential or change profile of software reuse. Only one survey illustrated that some metrics, such as CBO, may be relevant to reusability of a specific class [35]. As our study object is whole system, not a specific class or method, we decided *not* to use the above OO metrics in [9]. Instead, we chose to use the number of classes, the number of methods, in addition to NSLOC to measure the system size. Furthermore, we calculated the average complexity of all three systems using McCabes Cyclomatic complexity metrics [28]. To answer *RQ1.1*, we divided the total number of changes during the maintenance and evolution phases in JEF by the code size of the collected files. The same went for DCF and S&A.

We split the changes according to the date of the first release for the JEF, DCF, and S&A. All changes made after this date were regarded as maintenance and evolution changes. The changes made before or on this date were treated as development changes. We included only changes that were made during the maintenance and evolution phases for analysis.

To calculate the percentage covered by different types of changes and to answer *RQ1.2*, we classified the changes into different categories, according to the classification system proposed by [21] and used by [23]. The classification proposed in [21] is based on the actual maintenance activity performed. Given that each change description in the Rational ClearCase only describes what activities have been done to complete a software change, the activity-based classification of changes that is proposed in [21] was more suitable for our analysis and was therefore used. However, adaptive change is defined in [21] as "enhancements that add new system requirements". This definition does not specify whether these enhancements are related to environment or platform changes. We therefore decided to use the definition of adaptive change given in [42]. The definitions of change categories used in our study are also similar to those in [12, pp. 354–355], and are as follows:

- *Perfective:* encompass new or changed requirements, as well as optimizations.
- *Preventive:* changes related to restructuring and reengineering.
- *Corrective:* bug fixing.
- *Adaptive:* changes related to adapting to new platform or environments.

The first and second authors of the paper classified all the change descriptions separately. They then compared the results jointly. This resulted in 100% agreement. During the classification and comparison, we noticed that some of the change descriptions were labelled as "no changes" (meaning no changes were made to the code). We grouped "no changes" into the category "other". Some other changes were labelled as "initial review". (This means that changes were performed after a formal review of the code. However, the actual changes that were performed were not described in detail.) We classified "initial review" into the category "inspection", because changes in this category could not be classified exactly. After we had classified the changes, we calculated the change distribution of each type of change to the JEF, DCF and S&A.

To answer *RQ1.3*, we did a RCA [6] by showing all results of the *RQ1.1* to *RQ1.2* to a senior developer, who has followed all development and maintenance phases of the three systems, and asking him to give explanations. To avoid possible threats to validity, this developer was not informed of our research questions. We asked him to explain the results of *RQ1.1* to *RQ1.2* from the perspectives of *functionality*, *development practice*, *software complexity*, *age*, and *size*, because it has been claimed that these factors are determining factors of software maintenance [20]. Kemerer and Slaughter chose these five factors on the basis of data from the literature (i.e. [7]) and from case studies. Their analysis used a multivariate regression to determine the association between these five factors and the different maintenance types (e.g. enhancements, repairs, etc.), and they found these factors to be significant for software maintenance. To answer *RQ2.1*, we made a bar chart to show the distribution of different types of change over time (precise to the exact date). To answer *RQ2.2*, we did the same RCA as for *RQ1.3*.

5. Results and interpretations of the results

5.1. Software change profiles

We first compared the change densities of the selected files from three systems to answer *RQ1.1*. Detailed results are calculated

Table 4
Change density of the selected files of the three systems.

System	Total number of changes	Total number of classes	Changes per class	Total number of methods	Changes per method	Total number of NSLOC	Changes per NSLOC	Average complexity per file
JEF	398	526	0.76	3242	0.12	18362	0.022	1.78
DCF	2771	530	5.35	3662	0.76	20314	0.137	1.57
S&A	2094	776	2.70	5912	0.35	33018	0.063	1.70

using the JHawk tool [16] and shown in Table 4. The results illustrate that the selected files of the three systems have similar average McCabes Cyclomatic complexity. However, JEF has much lower change density, in terms of changes per NSLOC, changes per class, and changes per method, than DCF and S&A in maintenance and evolution phases.

We then compared the distributions of different change types of the three systems to answer *RQ1.2*. The detailed numbers (and percentage) of different types of change in the investigated systems are shown in Tables in Appendix C. The results reveal that:

- The *perfective* changes cover the highest percentage in the JEF, DCF, and S&A.
- Both S&A and DCF have a higher percentage of *preventive* changes than the JEF.
- S&A has a higher percentage of *adaptive* changes than the JEF and DCF.

The follow-up RCA provided explanations for the results for *RQ1.1* and *RQ1.2*, using the five factors by [20], as shown in Table 5.

To investigate *RQ1.1* and *RQ1.2* further, we also investigated the individual change profiles of the JEF, DCF, and S&A, according to Kemerer's five factors [20]. We also identified the component that covers the highest percentage of changes and the component with the highest change density. The observations and their interpretations are presented in Tables 6–8.

5.2. Software change profile over time

To investigate *RQ2.1*, we analyzed the change profile of different types of change of the three systems over time, as shown in Figs. 3–5. The RCAs of the observed change profiles are presented in Table 9, according to Kemerer's five factors [20].

The results from Figs. 3–5 reveal that:

- For the *JEF*, a few perfective and corrective changes were made after the first release was deployed (see stage S1 in Fig. 3). However, perfective changes dominated during this stage, with a peak of perfective changes (see P1 in Fig. 3) in June 2005. After stage S1, the number of corrective changes gradually increased,

Table 5
Results of root cause analysis for *RQ1.1 – RQ1.2*.

Observation	Factors and explanations
The JEF has lower change density than both DCF and S&A	*Development practice:* DCF had incomplete and poorly documented design, and higher time-to-market pressure (which required a large number of improvements over time), than JEF *Complexity:* DCF and S&A have more business logics than JEF. Business logic was changed often due to changed businesses
The *perfective* changes constitute the biggest category in JEF, DCF and S&A	*Functionality and development practice:* When the JEF was used by PDM, it was revealed that the Graphic User Interface (GUI) functionality did not satisfy client requirements. Thus, a lot of perfective changes had to be made DCF had time-to-market pressure, unclear requirements, and incomplete design at the early phases of implementation. This led to many perfective changes later S&A was first developed without involving the users. When the users got to see the application, many changes were made to requirements
Both S&A and DCF have higher percentages of *preventive* changes than the JEF	*Development practice:* JEF did not have high time-to-market pressure during development. That resulted in a good design and less need for refactoring *Development practice:* Time pressure and incomplete design of DCF led to some refactoring during implementation *Functionality:* S&A had more rules and business logic than the JEF and DCF, which led to some code cleanup and refactoring to ease later maintenance
S&A has more *adaptive* changes than JEF and DCF	*Development practice:* S&A is more closely coupled with JEF. When a new JEF release was deployed in February 2007, many derived adaptive changes have to be performed in S&A

Table 6
The results of root cause analysis for change profiles of the JEF.

Observation	Explanations
The total number of changes declined dramatically from releases 1 to 3	*Age:* Release 1 took two years to develop. Releases 2 and 3 took only two months each to develop. *Functionality and development practice:* The JEF is a reusable framework and is used by several applications. New features are not incorporated into the framework unless they will be used by at least by two different projects. The company is very careful when introducing changes to this framework, because changes to the API may affect many applications
Thirty percent of the changes made after the 2nd release and before the 3rd release were preventive	*Development practice:* During the implementation of the second release, a major refactoring was done to remove a cyclic dependency between security and session management components
After the third release, 30% of the changes were corrective, which was more than the percentages in releases 1 and 2	*Development practice:* A lot of the code in Release 2 was removed and replaced with third-party components. The increase in corrective changes was due to defects caused by these replacements
Most changes are in the JEFClient (29%) and JEFWorkBench (23%) components. Most changes in these components are perfective	*Size:* JEFClient constitutes the majority of the code *Functionality:* JEFClient and JEFWorkBench have GUI-related code. A lot changes were performed related to the GUI layout
The component with the highest change density is JEFIntegration	*Complexity:* JEFIntegration possess the most complex code *Functionality:* When the JEF framework was reused by different applications (PDM, DCF, etc.), new requirements for JEFIntegration emerged. The JEF was developed as a common framework to support GUI development, without knowing all the functionality that a framework may need at the early stage of the project

Table 7
The results of root cause analysis for change profiles of DCF.

Observation	Explanations
Most changes are in DCFClient (63%) and DCFCommon (41%) components. Most changes in these components are perfective	*Complexity and size:* These two components are the most complex and biggest components in DCF with respect to functionality
The component with the highest change density is DCFEJB	*Functionality:* This component has several configuration files and contains script and XML. These files need to be updated often, according to the context of deployment

Table 8
The results of root cause analysis for change profiles of S&A.

Observation	Explanations
Most changes are in SACommon (51%) and SAClient (48%) components. Most changes are perfective	*Size:* These two components are the largest components in S&A with respect to functionality
The component with the highest change density is SAClient	*Complexity:* SAClient is one of the most complex components in S&A with respect to functionality *Development practice:* The new JEF release impacts mainly the client part of S&A, which led to many adaptive changes in SAClient

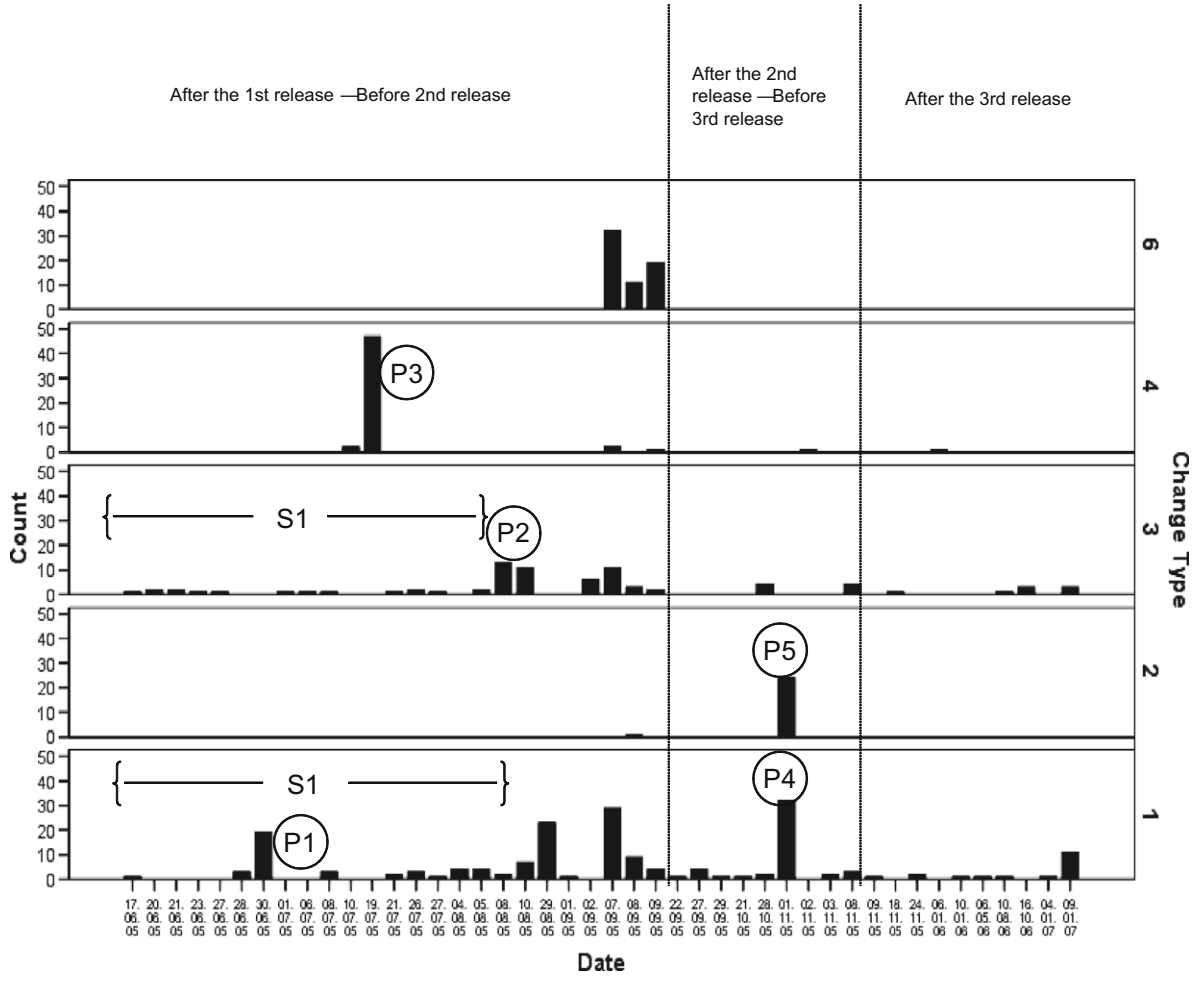

Fig. 3. JEF change profile: (1) perfective, (2) preventive, (3) corrective, (4) adaptive, (5) other, (6) inspection.

with a peak (see P2 in Fig. 3) one month before the release date of the second release; then it decreased. Specifically, there were many adaptive changes (in July 2005) between the end of the first release and the deployment of the second release (see P3 in Fig. 3). There was a peak of perfective changes (see P4 in Fig. 3) and a peak of preventive changes (see P5 in Fig. 3) in the middle between the second and third releases. However, few other changes were made during this period. After the third release, there were very few corrective and perfective changes.

– For *DCF*, the maintenance activity started with a few perfective changes after the first release (see stage S1 in Fig. 4). Then, a corrective change peak occurred (see P2 in Fig. 4). For preventive

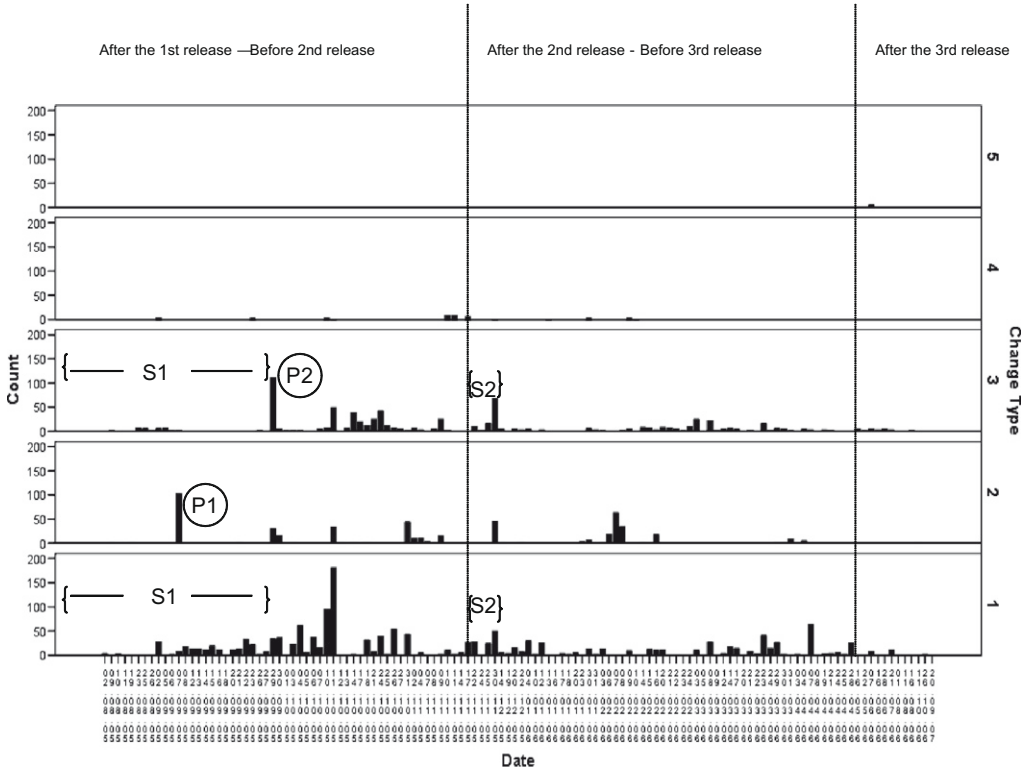

Fig. 4. DCF change profile: (1) perfective, (2) preventive, (3) corrective, (4) adaptive, (5) other.

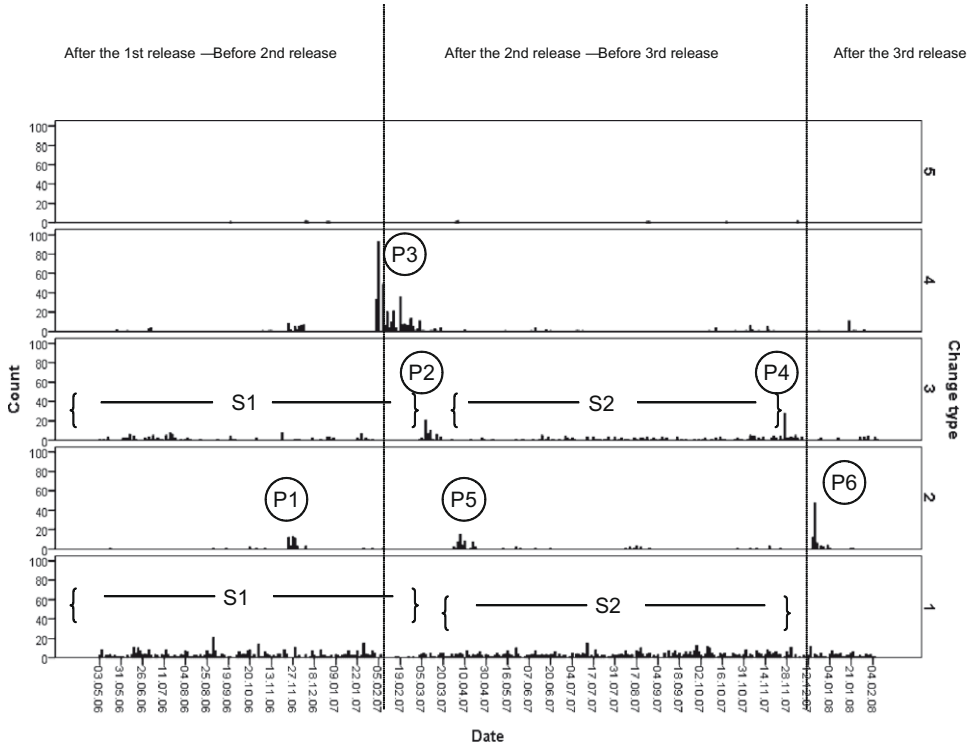

Fig. 5. S&A change profile: (1) perfective, (2) preventive, (3) corrective, (4) adaptive, (5) other.

changes, there was a peak (see P1 in Fig. 4) between the deployment of the first release and the second release in September 2005. After the deployment of the second release, the dominating changes were first perfective changes (see Stage S2 in Fig. 4), followed by corrective changes. After the third release, only few perfective and corrective changes were made.

– For *S&A*, the maintenance activity meant mainly perfective and corrective changes after the first release (see S1 in Fig. 5). However, it is obvious that there are much more perfective changes than corrective changes. Within stage S1, there was a peak of preventive changes (see P1 in Fig. 5). There were many adaptive changes around the release date of the second release (see P3 in

Table 9
The root cause analysis for *RQ2.1*.

Observation	Explanations
Many perfective changes were made to the JEF in June 2005 (see P1 in Fig. 3)	*Age:* The JEF was first used by PDM, before being reused by DCF and S&A (see Fig. 2). After PDM had used the JEF, several changes were made regarding the functionality of the JEF
Many corrective changes were made to the JEF in August 2005 (see P2 in Fig. 3)	*Age:* The corrective changes were made as a result of the intensive testing of the JEF before clients started to use the application
Many adaptive changes were made to the JEF in July 2005 (see P3 in Fig. 3)	*Development practice:* The adaptive changes were due to changes in the version control system. The company changed their version control system from PVCS to Rational ClearCase in the middle of the project. All the files in the PVCS had a Java comment, but when the company changed to Rational ClearCase, the Java comments in all the files were removed
Many perfective and preventive changes were made to the JEF in November 2005 (see P4 and P5 in Fig. 3)	*Development practice:* The perfective and preventive changes were due to the introduction of various third-party components
Many preventive and corrective changes were made to DCF in September 2005 (see P1 and P2 in Fig. 4).	*Development practice:* DCF was refactored in July 2005. The developers rewrote the whole DCFClient, which explains the large amount of preventive changes. Due to the refactoring, a new DCFClient was made. DCFClient then went through testing. That explains the high amount of corrective changes
Many preventive changes were made to S&A in November 2006, April 2007, and December 2007 (see P1, P5, and P6 in Fig. 5)	*Functionalities:* In November 2006, many files of S&A were refactored and then rewritten, due to changed requirements of some functionalities of the system *Development practice:* In April 2007 and December 2007, a lot of code clean-up and refactoring were made to facilitate code understanding and maintenance
A lot of adaptive changes were made to S&A in February and March 2007 (see P3 in Fig. 5)	*Development practice:* A new release of JEF was deployed in February 2007. All applications reusing JEF have to be changed to use the changed JEF version. Thus, around one month time was used in the S&A project to adapt the system to the new JEF release
Many corrective changes were made to S&A in March 2007 and November 2007 (see P2 and P4 in Fig. 5)	*Development practice:* In March 2007, a major security problem of the system was discovered. This led to changes in many database-related parts *Age:* Before the second release, the S&A were heavily tested. Thus, several defects were discovered, and led to intensive corrective activities in November 2007

Fig. 5) of S&A. Afterwards, the major maintenance activities were still perfective changes and corrective changes, with some intensive corrective changes happening at certain days (see P2 and P4 in Fig. 5). In addition, the preventive changes were also intensively implemented at certain dates (see P5 and P6 in Fig. 5).

6. Discussion

6.1. The overall distribution of different types of change: both reusable and non-reusable software

Our results reveal that *perfective changes* are the most common for both the reused framework and applications reusing the framework. Although slightly different definitions of change types have been used here, our results seem to support the observations of [25,18,11,36,30,23]. Several other studies [5,29,38] yielded different conclusions and showed that the majority of changes are either corrective or adaptive.

Our RCA shows that several factors, (i.e. complexity, functionality, development practice, age, and size) may determine the profile of software maintenance. From Tables 5–9 (counting the number of times each factor was chosen by the developer), we can see that the factors that affect maintenance the most in our case are *functionality* and *development practices*, followed closely by *complexity*. The factors that affect maintenance the least are *age* and *size*. In our case, the JEF received new requirements after it had been used by several applications, which led to many perfective changes being made during the maintenance and evolution phase. DCF faced high time-to-market pressure and had unclear requirements at the early stage of the development. For S&A, many perfective changes were made after new users saw the system and introduced new requirements. Without these evolving requirements, perfective changes might not have dominated in the systems that we investigated.

The study of Schach et al. [38] investigated three systems and observed that the dominant changes in their investigated systems are corrective, rather than perfective as in [25]. They explained the large amount of corrective changes by appeal to different programming domains, programming languages, and perspectives on study design. In our study, the dominant type of change was perfective instead of corrective. This is despite the fact that our study had a similar design to study [38]. We also investigated systems that were programmed in object-oriented languages. The only difference in our investigated systems from those in [38] was the programming domains. The products studied in [38] include one commercial real-time product, the Linux kernel, and GCC. Our investigated systems and those systems studied in [25] are basically business and data processing software, which often face new or changed requirements from clients due to their changing business needs. This difference may partially explains the fact that perfective changes dominated in the software studied in [25] and our study, and other types of changes dominated in the products examined in [38]. In addition to the application domain, our results illustrate that software change profiles can be dramatically influenced by several other factors and unexpected events. For example, the new release of JEF in February 2007 led to more than 20% changes of S&A and therefore heavily changed the distributions of different change types of the S&A.

Although results from our study support the conclusion of [25], that is, perfective changes dominate in software maintenance and evolution, generalizing this conclusion to other software system needs to be cautious. However, as noticed by Schach et al. [38], few textbooks (only one of the three top-selling ones [34,42,37]) illustrated that the results of [25] cannot be extrapolated to all types of software.

6.2. The distribution of different types of change: comparison between reusable vs. non-reusable software

With respect to differences in change density between the reused framework and applications which are reusing it, our results show that the reused framework indeed has a lower change density than the two applications reusing it. These results support

conclusions from studies [13,2,30,40]. The RCA revealed that developers in the company were cautious about making changes to the JEF, because the changes may affect existing applications. For example, a new release of JEF caused one month time of the S&A team to adapt the affected system to the modified JEF libraries. The change density of the reused framework may have been reduced as a result of this concern.

Another observation of our study is that both the reused framework and applications follow the so-called "80/20" rules [39,20], i.e. about 80% of all work is caused by only 20% of all components. In our case, one or two components in each system covered most of the changes, such as JEFClient and JEFWorkBench in JEF; DCFClient and DCFCommon in DCF; SACommon and SAClient in S&A. The components that require the most changes are usually the most complex, largest, or the ones involving many GUIs.

6.3. Software change profile over time

It is important to estimate the change profile of a software system in order to arrange staff expertise, tools, and business strategies properly [4]. Compared with previous studies on software change profiles [14,5,4], the change profiles of all our investigated systems support the simple/versioned stage model proposed by [4]. As we did not measure user-support activities; our study is not comparable with [5]. The evolution pattern proposed in [14] does not fit our data, because the first stages of the software changes of our investigated systems were not centered on corrective modifications.

For all the three systems, more perfective changes than corrective changes were made at the start of evolution and maintenance. Thus, the change profiles of the three systems are in line with the software evolution lifecycle proposed by [4]: that is, a stage to extend the software's capabilities and functionalities to meet user needs will follow the initial development, then a stage in which only minor defect repairs are made, and finally a phase-out stage. In addition, we noticed that the JEF arrived at the stage in which minor defect repairs are made faster than DCF and S&A. Very few changes occurred after the second release of the JEF, except a peak of perfective changes and a peak of adaptive changes. By contrast, DCF and S&A still followed Bennett's stage model, where many perfective changes were made, followed by corrective changes after the second release. In another word, the reused software goes to "code decay" [10] or "software rot" [45] more quickly than applications that reusing it.

The lower change rate can be explained by more sturdy design and quality checks, especially for shared class libraries, as documented by StatoilHydro's internal reuse guidelines. The many application clients of a reused class will also lead to prompt failure reports and quick bug fixes (corrective maintenance), i.e. fast stabilization ("maturization") after new releases. Furthermore, reusable software represents stable domain abstractions from the start, i.e. low change rates, with presumably modest evolution afterwards. However, behind such abstractions and their architectural layers may lie years of fumbling to get it "right" - also at StatoilHydro. Johnson and Foote write in [17] that useful abstractions are usually designed from the bottom up; i.e. they are "discovered" not invented (implying pre-designed from scratch).

6.4. Insights on the improvement of software reuse

Our results show both benefits and challenges of software reuse with respect to software evolution and maintenance. Reusable software may be more stable and need less maintenance effort, if the context of reuse can be predicted accurately. In addition, it is important to have a well-designed architecture to reduce the complexity of reusable software. For the long-term evolution and maintenance of reusable software, our results indicate that developers understanding reusable software well must be retained in the organization for a while after initial development and a new release is deployed. Such action is necessary because the software may experience a stage in which its capabilities and functionality are extended to meet user needs, which will require making many major changes, after the initial deployment. As proposed in [4], staff expertise is critical during the initial development of reusable software and when its capabilities and functionalities are being extended to meet user needs. Once the reusable software is stable and is at a stage in which only minor defect repairs are required, little staff expertise is needed because no dramatic change is welcome after the software has been reused by many applications.

6.5. Methodologies issues of our study and the software changes studies in general

In our study, we counted each check-in entry in the ClearCase system as one change. However, we did not measure the granularities of changes. One change may just need to modify one or two lines of code, while another change may need to modify vast amounts of lines of code. Thus, results of our study can only reflect the number of change entry in the system log without being able to reflect the effort used or code size involved. Counting the number of lines added, modified, deleted, or "touched file size by changes [10]" may partially solve this issue. However, many lines of code can be added or deleted using cut-and-copy commands within one second. The number of touched file size by each change cannot precisely reflect the actual effort to implement the change. Unless the effort to implement a specific change is logged and analyzed, it is difficult to predict the possible future maintenance and evolution effort of a software system from the number of changes or the type of changes are. However, few commercial software configuration management tools provide functionalities to recode the change effort.

6.6. Possible threats to validity

We here discuss possible threats to validity in our case study, using the definitions provided by [46]:

6.6.1. Construct validity

The definitions of different types of change used in our study are slightly different from those used in some previous studies, as was discussed in Section 2. Thus, direct comparisons of our results with previous studies need to be discussed case by case. RCA is often performed on each change. One possible threat to construct validity is that we performed our RCA on a subset of all changes. Given that we did not perform a detailed analysis of each change, we may have missed important details. However, in StatoilHydro ASA, several of the developers who are involved in the project are external consultants. When these consultants completed their work in the project, they left. This made it difficult for us to trace all changes back to each developer. Therefore, we did not have the resources to perform a RCA of each change. Another possible threat to construct validity is that we asked only one senior developer about the cause of the changes during the RCA. In addition, the same developer was asked for RCA in a parallel study [15], which focuses on software defects in the same systems. Possible correlations between the two studies may influence the answers from the developer. We eliminated this validity threat by not informing the developer any detailed research questions of both studies. Ideally, RCA with other developers can further verify or falsify explanations made by this developer. However, as mentioned above, external consultants left the project after their work

in a project is done. Thus, we could not find other people who knew the details of all the three systems that we investigated.

6.6.2. External validity

The entire data set was taken from one company. The object of study was a class framework and only two applications. Further similar studies need to be performed in different contexts and organizations in order to generalize our results.

6.6.3. Internal validity

All of the software changes that we investigated were classified manually by us. The first and the second author of the paper classified all the changes separately and then cross-validated the results. This was to enhance the validity of the data. A possible threat is whether the randomly selected subset files are representative of the whole system. We compared the relative code volume (classes per file, methods per file, and NSLOC per file) and average complexity of the selected files with all files in the three systems. The detailed data are shown in Appendix D and illustrate that our selected files are representative of all files in all the three corresponding systems.

6.6.4. Conclusion validity

This study is an explorative case study without any testing of hypotheses. Thus, the threat to conclusion validity is low.

7. Conclusion and future work

Few published empirical studies have characterized and compared the software changes made to a reusable framework with those made to a non-reusable application from a longitudinal perspective. We have presented the results from a case study of software changes that were performed on a reusable class framework and two applications reusing it. We studied the change density, the distributions of different types of change, and their properties over time. Our results contribute to a deeper understanding of software change over time and the relation between software change and software reuse. We conclude that:

- Our results support previous findings that reusable software is more stable than non-reusable software. One factor that contributes to reduce of the number of future changes of reusable software is its good initial design. Another factor is the concern of ripple effects of the changes in reused software to software reusing it.
- The change profile of the systems that we investigated during the maintenance and evolution phases usually goes as follows: initial development, followed by a stage in which the system's capabilities and functionality are extended to meet user needs, then a stage in which only minor defect repairs are made, and finally a phase-out stage. Reused software goes from extending the capabilities and functionality to minor defect repairs much faster then non-reusable software.
- The factors that affect maintenance the most in our case are software functionality, development practices, and software complexity. Other factors, such as age and size, also need to be considered when predicting software maintenance effort and when performing and presenting studies on software maintenance. However, further studies over a spectrum of application domains are required if their precise role and impact are to be determined.

We often end up with system architecture with (1) some tailored application functionality and logic at the top (i.e. towards end-users), (2) domain-specific layers in the middle, and (3) highly reusable and basic APIs (kernel and middleware functionality) at the bottom. However, the "middle" and "bottom" layers will constantly try to "eat" their way upwards as a result of a dynamic software marketplace. This "erosion from below" is challenging the profitable top layers that must be continuously upgraded and restructured (partly perfective and preventive maintenance). We can only observe some trends along these lines in our studied software, so this largely remains a topic for future research.

In addition, this study reports on only three systems in one company over three years and the results are exploratory. It is our intention to collect data on software change from more companies to validate our conclusions and to build a model to predict software maintenance and evolution activities and effort.

Acknowledgements

This work was financed in part by the Norwegian Research Council for the SEVO (Software EVOlution in Component-Based Software Engineering) project [41] with Contract Number 159916/V30. We would like to thank StatoilHydro ASA for the opportunity for involving us in their reuse projects and senior developer Thor André Aresvik for valuable comments.

Appendix A. Detailed information of the related studies

Table A1. The design and observations of the related studies.

Study description	Other observations of the study
A questionnaire-based survey that collected data from 69 systems, which were developed using different programming languages, e.g. Cobol, Assembler, Fortran, etc. [25]	User demands for enhancements and extensions constitute the most important problem area with respect to management
A case study that investigated change requests collected for two years in a Canadian financial institute [1] Used the same definitions as [25] for corrective, adaptive, and perfective changes Analyzed 2152 change requests	Maintenance team in 1989 spent 64% of their time doing maintenance work (e.g. optimization and adding new functionality) other than correcting defects and errors
A survey conducted in the Management Information System (MIS) department in nine different application domains in Hong Kong	In Hong Kong, 66% of the total software life cycle effort was spent on software maintenance

(*continued on next page*)

Table A1 (*continued*)

Study description	Other observations of the study
1000 questionnaires were sent out and about 50 responses were received [47]	The most cited maintenance problems were staff turnover, poor documentation, and changing user requirements
A structured interview with managers and maintainers in a computer department of a large Norwegian telecom organization in 1990–1991 (study 1) and 1992–1993 (study 2) [18] Systems were developed using either Cobol or Fourth Generation languages	If the amount of corrective work is calculated on the basis of interviews solely with managers, it will be as twice as much as the actual work reported in logs (i.e. the amount of corrective work may be exaggerated in interviews)
Studied 10 projects conducted in the Flight Dynamic Division (FDD) in NASA's Goddard Space Flight Center. The FDD maintains over 100 software systems totalling about 4.5 million lines of code. 85% of the systems are written in FORTRAN, 10% in Ada, and 5% in other languages [3]	Error corrections are small isolated changes, while enhancements are larger changes to the functionality of the system More effort is spent on isolation activities in correcting code than when enhancing it
A case study investigated the change of maintenance requests during the lifecycle of a large software application (written in SQL) [5] Analyzed 654 change and maintenance requests	User support reaches its peak in the 4th month (first stage). Repair reaches its peak in the 13th and 14th months (second stage), while enhancement is the dominant factor in the third stage (25th month)
A survey carried out in financial organizations in Portugal Data was collected from 20 project managers [43]	3% of the respondents considered the software maintenance process to be very efficient, while 70% considered that efficiency is very low
An Ada system of the NASA Goddard Space Flight Center [11] Analyzed 453 non-defect changes	Changes related to optimizations require the most effort to isolate, while planned enhancements require the most effort to implement
A subsystem that contains two million lines of source code [29] Analyzed 33171 modification requests	Corrective changes tend to be the most difficult, while adaptive changes are difficult only if they are large. Inspection changes are perceived as the easiest
A case study on reengineering a people-tracking subsystem of an automated surveillance system, which was written in C++ and had 41 KLOC [36] Analyzed the distribution of maintenance effort during the whole maintenance phase	The effort required to adapt the system was high, because the software needed to be ported to a different platform
Examined three software products: – A real-time product written in a combination of assembly language and C. Data of 138 modified versions were collected – The Linux kernel. Data from 60 modified versions were collected – GCC (GNU Compiler Collection). Data from 15 versions were collected [38]	All three maintenance categories were statistically very highly significantly different from the results of [25] Corrective maintenance was more than three times the level of the results of [25]
Four releases of a telecommunication system written in Erlang, C, Java, and Perl [31] Analyzed 187 change requests	There is no significant difference between reused and non-reused components in the number of change requests per KSLOC
Web-based Java application, consisting of 239 classes and 127 JSP files [23] Based on Swanson's definition [44] and Kitchenham's ontology [21] Analyzed 93 fault reports	Maintenance effort of Java application is similar to the distribution in previous non object-oriented and non web-based applications

Appendix B. Detailed information of the investigated systems

The JEF is designed on the basis of a technical architecture for all J2EE systems in the company. This architecture has four logical layers. The following presentation describes the layers from top to bottom:

(1) *Presentation:* Responsible for displaying information to the end user and to interpret end-user input.
(2) *Process:* Provides support for the intended tasks of the software and configures the domain objects.
(3) *Domain:* Responsible for representing the concepts of the business and information about the business and business rules. This layer is the heart of the system.
(4) *Infrastructure:* Provides general technical services such as transactions, messaging, and persistence.

The components in this framework are built from a combination of commercial-off-the-shelf (COTS) components, open source system (OSS) components, and some code that was developed in-house. Table B1 gives information about the JEF, using Kitchenham's ontology of software maintenance as a basis [21].

DCF is used mainly for storing documents. It imposes a certain structure on the documents stored in the application and relies on the assumption that the core part of the documents is based on cargo (load) and deal (contract agreement) data, as well as auxiliary documents that pertain to this information. DCF is meant to replace the current handling of cargo files, which are physical folders that contain printouts of documents that pertain to a particular cargo or deal. A cargo file is a container for working documents that are related to a deal or cargo, within operational processes, used by all parties in the oil sales, trading, and supply strategy plan of the company. There are also three releases of the DCF application. Table B2 gives information about DCF.

Table B1. Background information for the JEF.

Ontology	Description
Application domain	Java technical framework for developing enterprise applications
Product age	3.5 years
Product maturity	Adolescence
Product composition	Consists of seven separate components, which can be applied separately or together when developing applications
Product and artefact quality	Well-defined requirement and design document
Development technology	– J2EE (Java 2 Enterprise Edition) – SPRING framework
Paradigm	– Object-oriented paradigm (Java) – Design patterns – Partially open-source development
Maintenance organization process	– Software development plan – Software configuration management tool

Table B2. Background information for DCF.

Ontology	Description
Application domain	A document storage application to manage cargo files. A cargo file is a container for working documents that are related to a deal or cargo, within operational processes, used by all parties in the company
Product age	3.5 years
Product maturity	Senility (legacy)
Product composition	Eight components built in-house, Biztalk, MessageManager, and Meridio
Product and artifact quality	Poor requirement and design document in the beginning
Development technology	– J2EE (Java 2 Enterprise Edition) – SPRING framework
Paradigm	– Object-oriented paradigm (Java) – Design patterns
Maintenance organization process	– Software development plan – Software configuration management tool

S&A is an application for controlling business processes and carrying them out more efficiently through common business principles within lift and cargo planning. Lift planning is based on a lifting program, which is the function area for generating an overview of the cargoes that are scheduled to be lifted. The lifting program operates on a long-term basis (e.g. 1–12 months), and generates tentative cargoes mainly on the basis of closing stock and predictions about production. The lifting program is distributed to the partners so that they can plan how they will handle the lifting of their stock. The cargo planning and shipment covers activities to accomplish the lifting. Input to the process is the lifting program. While carrying out the process, users will enter detailed information about the cargo on the basis of document instruction from partners and perform short-term planning on the basis of pier capacity and storage capacity. After loading, sailing telex and cargo documents are issued. Then the cargo is closed and verified. The S&A application allows the operators to carry out "what-if" analysis on shipments that are to be loaded at terminals and offshore. Given that the current system ("SPORT") is not able to handle complex agreements (i.e. the mixing of different qualities of oil within the same shipment), it allows the transfer and entry of related data, which is currently often done manually, to be digitized and automated. The main goal of the S&A application is to replace some of the current processes/systems, in addition to providing new functionality. The S&A application has also three releases. Table B3 gives information about S&A.

Table B3. Background information for S&A.

Ontology	Description
Application domain	An application for controlling and performing business processes more efficiently through common business principles within lift and cargo planning
Product age	2.5 years
Product maturity	Senility (legacy)
Product composition	Seven components built in-house
Product and artifact quality	Well-defined requirement and design document
Development technology	– J2EE (Java 2 Enterprise Edition) – SPRING framework
Paradigm	– Object-oriented paradigm (Java) – Design patterns
Maintenance organization process	– Software development plan – Software configuration management tool

Appendix C. Detailed results of research question *RQ1*

Table C1. JEF releases and the number and percentage of changes.

JEF	Maintenance and evolution			
	After the first release and before second release	After the third release and before third release	After third release	In total
Perfective	115 (39%)	46 (58%)	18 (67%)	179 (45%)
Corrective	62 (21%)	8 (10%)	8 (30%)	78 (20%)
Preventive	1 (0.3%)	24 (30%)	0 (0%)	25 (6%)
Adaptive	52 (18%)	1 (1%)	1 (4%)	54 (14%)
Inspection	62 (21%)	0 (0%)	0 (0%)	62 (16%)
Other	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Total	292	79	27	398

Table C2. DCF releases and the number and percentage of changes.

DCF	Maintenance and evolution			
	After the first release and before second release	After the third release and before third release	After third release	In total
Perfective	910 (56%)	580 (53%)	25 (46%)	1515 (55%)
Corrective	421 (26%)	282 (26%)	24 (44%)	727 (26%)
Preventive	273 (17%)	213 (20%)	1 (2%)	487 (18%)
Adaptive	24 (2%)	13 (1%)	0 (0%)	37 (0.1%)
Inspection	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Other	0 (0%)	0 (0%)	5 (9%)	5 (0%)
Total	1628	1088	55	2771

Table C3. S&A releases and the number and percentage of changes.

S&A	Maintenance and evolution			
	After the first release and before second release	After the third release and before third release	After third release	In total
Perfective	399 (53%)	577 (50%)	88 (44%)	1064 (51%)
Corrective	113 (15%)	230 (20%)	20 (10%)	363 (17%)
Preventive	54 (7%)	71 (6%)	78 (39%)	203 (10%)
Adaptive	175 (24%)	259 (23%)	16 (8%)	450 (22%)
Inspection	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Other	6 (1%)	8 (1%)	0 (0%)	14 (1%)
Total	747	1145	202	2094

Appendix D. Profiles of selected files vs. all files in the three systems

Table D1. Size and complexity of selected files vs. all files in the three systems.

Files	Total number of files	Classes per file	Methods per file	NSLOC per file	Average complexity per file
All JEF files	634	1.2	7.2	40.6	1.73
Selected JEF files	442	1.2	7.33	41.5	1.78
All DCF files	694	1.05	7.6	42.3	1.61
Selected DCF files	501	1.06	7.3	40.5	1.57
All S&A files	2823	1.09	8.5	48.3	1.65
Selected S&A files	700	1.11	8.46	47.2	1.70

References

[1] A. Abran, H. Nguyenkim, Analysis of maintenance work categories through measurement, in: Proceedings of the IEEE Conference on Software Maintenance, IEEE Computer Society Press, Sorrento, Italy, 1991, pp. 104–113.
[2] H. Algestam, M. Offesson, L. Lundberg, Using components to increase maintainability in a large telecommunication system, in: Proceedings of the Ninth IEEE Asia-Pacific Software Engineering Conference, IEEE Computer Society Press, Gold Coast, Australia, 2002, pp. 65–73.
[3] V. Basili et al., Understanding and predicting the process of software maintenance releases, in: Proceedings of the 18th International Conference on Software Engineering, IEEE Computer Society Press, Berlin, Germany, 1996, pp. 464–474.
[4] V.T. Rajlich, K.H. Bennett, A staged model for the software life cycle, IEEE Computer 33 (2000) 66–71.
[5] E. Burch, H.J. Kung, Modeling software maintenance requests: a case study, in: Proceedings of the IEEE International Conference on Software Maintenance, IEEE Computer Society Press, Bari, Italy, 1997, pp. 40–47.
[6] D.N. Card, Learning from our mistakes with defect causal analysis, IEEE Software 15 (1998) 56–63.
[7] N. Chapin, Software maintenance: a different view, in: AFIPS Conference Proceedings, NCC, vol. 54, AFIPS Press, Reston, Virginia, 1985, pp. 328–331.
[8] N. Chapin et al., Types of software evolution and software maintenance, Journal of Software Maintenance and Evolution: Research and Practice 13 (2001) 3–30.
[9] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20(6) 476–493.
[10] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, A. Mockus, Does code decay? Assessing the evidence from change management data, IEEE Transactions on Software Engineering 27 (2001) 1–12.
[11] W.M. Evanco, Analyzing change effort in software during development, in: Proceedings of the 6th IEEE International Software Metrics Symposium, IEEE Computer Society Press, Boca Raton, Florida, 1999, pp. 179–188.
[12] N.E. Fenton, S.L. Pfleeger, Software Metrics, PWS Publishing Company, 1996.
[13] W.B. Frakes, An industrial study of reuse, quality, and productivity, Journal of Systems and Software 57 (2001) 99–106.
[14] D. Gefen, S.L. Schneberger, The non-homogeneous maintenance periods: a case study of software modifications, in: Proceedings of the IEEE International Conference on Software Maintenance, IEEE Computer Society Press, Monterey, California, 1996, pp. 131–141.
[15] A. Gupta, J. Li, R. Conradi, H. Rønneberg, E. Landre, A case study comparing defect profiles of a reused framework and of applications reusing the same framework, Journal of Empirical Software Engineering 14 (2009) 227–255.
[16] Java Metrics Calculator, <http://www.virtualmachinery.com/jhawkprod.htm>, 2009.
[17] Ralph E. Johnson, B. Foote, Designing reusable classes, Journal of Object-Oriented Programming 1 (3) (1988) 26–49.
[18] M. Jørgensen, The quality of questionnaire based software maintenance studies, ACM SIGSOFT – Software Engineering Notes 20 (1995) 71–73.
[19] T. Kamiya, S. Kusumoto, K. Inoue, Prediction of fault-proneness at early phase in object-oriented development, in: Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Saint-Malo, France, 1999, pp. 253–258.
[20] C.F. Kemerer, S.A. Slaughter, Determinants of software maintenance profiles: an empirical investigation, Journal of Software Maintenance 9 (1997) 235–251.
[21] B.A. Kitchenham et al., Towards an ontology of software maintenance, Journal of Software Maintenance: Research and Practice 11 (1999) 365–389.
[22] J. Krogstie, A. Jahr, D.I.K. Sjøberg, A longitudinal study of development and maintenance in Norway: report from the 2003 investigation, Information and Software Technology 48 (2006) 993–1005.
[23] M.G. Lee, T.L. Jefferson, An empirical study of software maintenance of a web-based java application, in: Proceedings of the 21st IEEE International Conference on Software Maintenance, IEEE Computer Society Press, Budapest, Hungary, 2005, pp. 571–576.
[24] M.M. Lehman, Programs, life cycles and laws of software evolution, Proceedings of Special Issue Software Engineering 68 (1980) 1060–1076.
[25] B.P. Lientz, E.B. Swanson, G.E. Tompkins, Characteristics of application software maintenance, Communications of the ACM 21 (1978) 466–471.
[26] W. Lim, Effect of reuse on quality, productivity and economics, IEEE Software 11 (1994) 23–30.
[27] Marketing Correlation, <http://www.macorr.com/ss_calculator.htm>, 2008.
[28] McCabe and Associates, McCabe Object-Oriented Tool User's Instructions, User Manual, 1994.
[29] A. Mockus, L.G. Votta, Identifying reasons for software changes using historical databases, in: Proceedings of the IEEE International Conference on Software Maintenance, IEEE Computer Society Press, San Jose, California, 2000, pp. 120–130.
[30] P. Mohagheghi, R. Conradi, O.M. Killi, H. Schwarz, An empirical study of software reuse vs. defect density and stability, in: Proceedings of the 26th IEEE International Conference on Software Engineering, IEEE Computer Society, Edinburgh, Scotland, 2004, pp. 282–292.
[31] P. Mohagheghi, R. Conradi, An empirical study of software change: origin, impact, and functional vs. non-functional requirements, in: Proceedings of the IEEE International Symposium on Empirical Software Engineering, IEEE Computer Society Press, Redondo Beach, Los Angeles, 2004, pp. 7–16.
[32] T.M. Pigoski, Practical Software Maintenance, Wiley Computer Publishing, 1997.
[33] M. Postema, J. Miller, M. Dick, Including practical software evolution in software engineering education, in: Proceeding of the 14th Conference on Software Engineering Education and Training, IEEE Computer Society Press, Charlotte, North Carolina, 2001, pp. 127–135.
[34] R.S. Pressman, Software Engineering, A Practitioner's Approach, McGraw-Hill, Boston, 2001.

[35] L. Rosenberg, in: Applying and Interpreting Object Oriented Metrics, Software Technology Conference, <http://www.stsc.hill.af.mil/crosstalk/1997/04/quality.asp>, 1998.
[36] M. Satpathy, N.T. Siebel, D. Rodríguez, Maintenance of object oriented systems through re-engineering: a case study, in: Proceedings of the 10th IEEE International Conference on Software Maintenance, IEEE Computer Society Press, Montreal, Canada, 2002, pp. 540–549.
[37] S.R. Schach, Object-Oriented and Classical Software Engineering, McGraw-Hill, Boston, 2002.
[38] S.R. Schach, B. Jin, L. Yu, G.Z. Heller, J. Offutt, Determining the distribution of maintenance categories: survey versus management, Journal of Empirical Software Engineering 8 (2003) 351–366.
[39] H. Schaefer, Metrics for optimal maintenance management, in: Proceedings Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, CA, 1985, pp. 114–119.
[40] W. Selby, Enabling reuse-based software development of large-scale systems, IEEE Transactions on Software Engineering 31 (2005) 495–510.
[41] The Software EVOlution (SEVO) Project, <http://www.idi.ntnu.no/grupper/su/sevo/>, 2004–2008.
[42] I. Sommerville, Software Engineering, Addison-Wesley, UK, 2004.
[43] M. Sousa, H. Moreira, A survey on the software maintenance process, in: Proceedings of the 14th IEEE International Conference on Software Maintenance, IEEE Computer Society Press, Bethesda, Maryland, 1998, pp. 268–274.
[44] E.B. Swanson, in: Proceedings of the Second IEEE International Conference on Software Engineering, IEEE Computer Society Press, San Francisco, California, 1976, pp. 492–497.
[45] Wikipedia on Software Rot, <http://en.wikipedia.org/wiki/Software_rot>, 2008.
[46] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering – An Introduction, Kluwer Academic Publishers, 2002.
[47] S. Yip, T. Lam, A software maintenance survey, in: Proceedings of the First IEEE Asia-Pacific Software Engineering Conference, IEEE Computer Society Press, Tokyo, Japan, 1994, pp. 70–79.
[48] P. Yu, T. Systa, H. Muller, Predicting fault-proneness using OO metrics: an industrial case study, in: Proceedings of 6th European Conference Software Maintenance and Reengineering, IEEE Computer Science Press, Budapest, Hungary, 2002, pp. 99–107.
[49] W. Zhang, S. Jarzabek, Reuse without compromising performance: industrial experience from RPG software product line for mobile devices, in: Proceedings of the 9th International Software Product Line Conference, Springer, Rennes, France, 2005, pp. 57–69.
[50] Y. Zhou, H. Leung, Empirical analysis of object-oriented design metrics for predicting high and low severity faults, IEEE Transactions on Software Engineering 32 (2006) 771–789.