A Case Study of The Class Firewall Regression Test Selection Technique on a Large Scale Distributed Software System

Mats Skoglund  
Dept. of Computer and Systems Sciences  
Stockholm University/Royal Institute of Technology  
Forum 100, 164 40 KISTA, SWEDEN  
matte@dsv.su.se  
+46-8 161536

Per Runeson  
Dept. of Communication Systems  
Lund University  
Box 118, 221 00 LUND, SWEDEN  
per.runeson@telecom.lth.se  
+46-46 2229325

Abstract

Regression testing is expensive and may consume much of organizations’ software development budgets. Thus, it is of interest to reduce the total time devoted to test execution by using test selection techniques. Many techniques have been proposed but few have been evaluated on real-world, large scale systems. In this paper we report on an empirical evaluation of using the class firewall regression test selection technique, in combination with scenario testing, on a large scale industrial software system using the Java byte code in the analysis. The study was performed on a large complex distributed software system in one of Sweden’s largest banks. Effects of using scenario testing together with regression test selection are reported. The results are that not all test cases were selected by the class firewall selection technique. Using scenario testing, where test cases are dependent, affects the number of test cases selected, as do the location and number of changes in the system.

1 Introduction

A new version of a software system should be re-tested before it is released to its users to assure that features that worked in the previous version still work in the new version. This process is often referred to as regression testing and is typically performed by re-executing previously constructed test cases on the new version [1]. Regression testing is expensive and may consume much of organizations’ software development budgets [15]. When regression testing is performed manually, the cost of executing the test cases may be one of the most expensive activities of the testing process. Thus, it is of interest to many organizations to reduce the total time devoted to the test execution. When only limited changes have been made to the software, a complete re-run of all test cases may be unnecessary and instead executing only a subset of the full test suite may suffice. To select test cases for re-executing is referred to as regression test selection [9].

Many techniques have been proposed that use information about the previous and current versions of the software to select only a part of the full test suite but still having a good probability of detecting faults in the system, e.g. [3, 5, 14, 4, 7, 12, 11]. An important property of a test reduction technique is that it is safe. A safe technique is a technique that under given circumstances selects test cases that will expose the same set of faults as the complete test suite would expose [13]. Using a safe technique, the testing cost will be reduced without reducing the test effectiveness. One safe regression test selecting technique proposed for object-oriented software is the class firewall technique [7, 16]. This technique assumes that only those classes of the program that have been changed and those classes possibly affected by the changes need to be re-tested.

The firewall technique identifies which parts of the system that need re-testing, without requiring any specific testing method to be used. This means that the technique may be applied with test cases for integration testing of subsystems as well as with test cases for complete system testing. The technique may also be used together with different testing methods, such as with unit tests written in the same programming languages as the system, protocol testing using message passing or use-case testing via the user interface. However, properties of the test cases may impact on the overall effectiveness of using a specific regression test selection technique. For example, if a test case A re-
quires another test case, \( B \), to be completed before \( A \) is executed, then if a selection technique selects only test case \( A \) but not \( B \), then \( B \) still must be executed, leading to a larger test suite.

Some regression test selection techniques may require access to the source code to perform various analyses on. This may lead to difficulties if, for example, several different programming languages have been used for developing the system, e.g. when using Components Off The Shelf (COTS). Then different techniques may be required for each different programming language used, which may be both time consuming and sometimes even impossible. However, in environments such as e.g. the Microsoft .NET environment, Visual Basic, C# and C++ all produce the same intermediate code type, the Microsoft Intermediate Language (MSIL) [6]. Thus, by analyzing the intermediate code instead of the source code, the whole system can be analyzed by using only one technique.

In this paper we report on an empirical evaluation of using the class firewall regression test selection technique, in combination with scenario testing, on a large scale industrial software system using the Java byte code in the analysis. The study was performed on a large complex distributed software system in one of Sweden’s largest banks. The system is written in J2EE\(^1\) and has many internal and external dependencies to other environments such as mainframes and special hardware.

The research question for this flexible study is “How may test suite reduction be performed together with a scenario based testing technique on large scale software”? The starting point is the class firewall regression test selection technique.

The outline for the remainder of this paper is as follows. In Section 2 background to the techniques used are presented. Section 3 contains related work and in Section 4 the case study subject is described. The data collection and analysis procedures are outlined in Section 5. Section 6 contains the results, Section 7 discusses threats to validity of the study and Section 8 concludes.

## 2 Background

This section gives a short introduction to the class firewall regression test selection technique, scenario testing and further motivation for using intermediate code in regression test selection.

---

\(^1\)Java Enterprise Edition: http://java.sun.com/j2ee
agram for the system. Another is that the test cases need to be traced to collect information of which classes are used by each test case. Also, the changed classes and the classes dependent on the changed classes, i.e. the changed classes’ firewalls should be identified. Finally, the test cases using the classes in the firewall should be selected and executed.

2.2 Scenario Testing

A system’s behaviour may be described by a set of use-cases that specify how it may be used from the user perspective and thus use-cases may also be used for specification based testing [2]. Use-cases are usually created during the development process to gather and specify requirements of the system. While use-cases describe a whole set of scenarios for a system, a specific scenario is a more detailed description of the sequences on using the system [10], [2, p. 225]. Depending on e.g. the system under test and on the strategy used to create the scenarios, there are numerous ways of defining and using scenarios in a testing setting. For example, a system may be tested using only one single scenario that contains a series of steps performed one after each other and where each step requires the previous step to be successfully completed.

Scenarios are usually specified informally in free text [15] and thus probably executed manually. A large system may require much effort to re-execute the scenarios during regression testing and thus reducing the number of scenarios using test selection techniques is relevant.

2.3 Intermediate Code for Testing

An issue with large scale industrial software systems is that a high number of dependencies to other systems may lead to very complex execution and development environments. A system running on dedicated servers, where many adjustments have been made to suit the organization’s purposes, may require a big effort to replicate in order to e.g. create new testing and analysis environments. Also, with large software systems it may be infeasible, due to configuration and hardware costs, to create development environments for the engineers that can contain the whole system at once. Instead, each engineer may only have immediate access to the parts of the system that she is currently working on and if other parts are needed they will be loaded on demand. If a special development or testing environment must be created from scratch for the test selection process in such a complex environment, the process of regression test selection may be both cumbersome and prone to error. However, if an environment for e.g. functional testing already exists, then analyzing the intermediate code from this testing environment may be less costly than creating a new analysis environment. Using an existing test environment may also be less prone to errors since the analysis is performed on the same artifact, the intermediate code, as actually used for executing the tests and thus problems related to for example versioning cannot occur.

3 Related Work

The class firewall regression test selection technique was presented 1995 by Kung et. al [7]. They also provided an example of the use of the technique. The example was based on the InterViews library containing approximately 140 classes and 400 relationships and is written in C++. They concluded that automatic support is needed to identify and keep track of the many relationships that exist in an object-oriented program. Their example is focused on the use of the class firewall technique to select the order in which test cases should be executed to save effort in creating drivers and stubs. However, neither test cases nor testing methods are provided in the example and thus no evaluation of the combination of the selection technique together with a testing method was performed.

Orso et. al. [9] propose a regression test selection technique for Java programs that can be applied to large scale software. The technique is a combination of different techniques to increase the effectiveness when used on large software systems. The technique is also empirically evaluated on two small and one larger stand-alone Java applications. In one of the subjects studied, a 62.5% saving of testing time was found. However, it is unclear from the paper which testing method was used in the evaluations.

Hsia et. al [4] evaluated the class firewall selection technique on an ATM simulation program written in C++. The test cases may be described as scenario tests where each test case tested one specific part of the software, such as “Input invalid user ID, exit”, “Test the deposit of money, invalid input” and “Test the withdraw of money, invalid input”. The technique was also evaluated using scenarios on a drawing program written in C++. It is concluded that the class firewall technique may save time since not all scenarios are selected for regression testing. Another conclusion is that it is expected that the approach will save approximately 67% for the software revalidation. These experiments were however conducted in a small scale with systems consisting of 13 and 26 classes respectively and not further validated on larger systems.

Jang et. al describes a test selection technique for
C++ [5]. Their technique is a modification of the original class firewall technique where more detailed information is used when constructing the firewalls. The technique is applied also on the rather small InterViews drawing application. They conclude that their more detailed technique is only applicable to systems of small sizes due to the high analysis cost, but no empirical justification for this is provided.

Harrold et al proposes a safe regression test selection technique especially for Java [3]. The technique is control-flow based and uses type information and information regarding the control flow in the program to compute which parts need re-testing after a change. The empirical evaluation of the technique is performed on four different software subjects. The subjects sizes are measured in number of methods, and the subjects consist of 109, 168, 185 and 3495 methods respectively. They used both existing test cases and constructed new test cases for the subjects. For the largest system it is reported that 189 test cases are created, using the user interface of the system. These test cases may be considered as scenario tests. The application tested is a text editing software named JEdit\(^2\). For 6 of the 11 versions analyzed more than 90\% of the test cases were selected for retesting and for the remaining 5 versions less than 30\% of the test cases were selected. For 3 of the versions only minor modifications had been performed and thus only a few methods were affected, leading to that a small number of test cases were selected. They also conclude that other properties, apart from the regression test selection technique, affects the number of test cases selected. Among those properties are the number of changes performed, the frequency of regression testing, the location of the changes and the characteristics of the software. Their study was however not performed using class firewall as the selection technique.

The class firewall regression test selection technique has not, to our knowledge, been evaluated in combination with scenario testing on a large scale industrial system prior to this study.

4 Study Object

The study object of this case study is a large distributed component-based J2EE system. The system is owned by Swedbank (FöreningsSparbanken AB), one of Sweden’s largest banks. The system is used both internally and externally by employees, customers and business partners. The system consists of several web based applications, all using core functionality provided in a common component infrastructure framework. The system consists of software developed in-house as well as third-party components. The system consists of more than 1 200 000 lines of code compiled into approximately 27 000 classes.

The system has many internal and external dependencies leading to a very complex system that is difficult to test. There are dependencies within the system via the common infrastructure, such as to logging and authentication services. One important driving force in the architecture is to create a large degree of reuse in the development of applications, which means that different applications use common components, leading to dependencies between applications. There are also dependencies to mainframe systems and other environments. These systems are used by many applications both in the studied system and from other systems within the bank. The system also communicates with systems from external providers such as national and international money transferring systems. Due to e.g. performance reasons, special hardware, such as load balancing hardware, is also used by specific applications.

The system is component-oriented and each component executes on one of four different application servers, depending on the purpose of the component. The purpose that maps to an individual application server may be either of:

1. internal use, components for applications mostly used by employees and partners
2. external use, applications and components mostly used by customers
3. core components, infrastructure components and common components used by several applications
4. special hardware, applications requiring special hardware

Major new versions of the system are released 3-4 times per year and additional bug fixes are incorporated and released 2-3 times between major releases. Full regression testing is performed only for major releases while for minor releases, partial regression testing is performed. The parts are selected based on judgments by developers and test engineers. Only a few faults, less than 30, are detected during the regression testing process and even fewer, less than 10, have been detected after a new version has been released. Each major release of the system is focused on either the applications or the infrastructure. For application releases the organization focuses on releasing new and

\(^2\)http://www.jedit.org/
changed applications, while infrastructure releases are focused on new and changed infrastructure, such as logging or authentication services. An application release should not contain any new or changed infrastructure parts and infrastructure releases should not contain any new or major changed applications.

The regression testing is performed using scenario based testing. New applications and applications that have undergone a major change are regression tested within the development project organizations, while existing applications are regression tested by a special regression testing team. Since all applications in the system use a common set of components, a change in a common component require all applications using the changed component to be tested. These regression tests are performed by the test engineers who follow scenario procedures for each application where various tasks are performed in the application, verifying its behaviour.

5 Data Collection and Analysis

Using the system and its environment described in the previous section we conducted data collection and analysis to study regression test selection techniques in combination with scenario testing.

Manual processing was infeasible due to the size of the system and its complexity and thus it was necessary to find and develop tools for collecting data about the system to be able to perform regression test selection on the study subject. Tools and methods were needed for 1) extracting dependency information about all classes in the system, 2) compute differences between versions of the system to identify changes, and 3) identifying which test cases use which classes.

5.1 Extracting Dependency Information

The tool we used for extracting and collecting dependency information was a tool called Dependency Finder\(^3\). It operates on compiled class files and may extract dependency information into a XML-based format.

5.2 Identifying Changes

The information in the configuration management system could not be used to calculate differences between versions, since the versioning information is stored on a more abstract "component" level and not on individual class level. Instead we produced MD5\(^4\) signatures for each compiled class file. An MD5 signature is computed from the contents of the class file and it contains a 128-bit "fingerprint" of the file. If the file is changed and a new signature is computed for the changed file, the new signature has a very high probability of being different from the first signature. Thus, signatures may be used for detecting whether one version of a compiled class file is different from another and needs re-testing. We developed an own tool for calculating signatures for all classes in the system.

5.3 Coverage Analysis

If a class is changed the test cases that use the changed class should be included in the regression test suite. To determine which test cases used which classes, coverage analysis was conducted. This was performed by instrumentation of the Java byte code and run-time collection of coverage data during the original testing.

We used the freely available tool Emma\(^5\) for coverage analysis. Both class files packaged in jar-files and individual class files were instrumented. The data collection process requires that the tool is available in the Java virtual machine during runtime. Thus the tool was installed in all four application servers in the system.

Not every test case executes code on the application servers. For example test cases that use client side Java scripts to test that help windows are displayed correctly. Since these test cases cannot be selected in the test selection process they are excluded from the analysis, unless they are relevant for other purposes in the study. For test cases requiring some other test case to execute prior to it, we could not identify the exact coverage for each of the test cases individually due to limitations in the coverage analysis tool. Thus, for a test case where no more code was covered compared to the previous test case, it could not be determined whether the current test case executed code that was already covered by the previous test case or if no code was executed at all. These test cases are also excluded from parts of the analysis. There were a total number of 69 test cases of which 51 were included in the analysis and thus we excluded 18 test cases.

5.4 Subjects Tested

Three applications in the system with different characteristics were selected to execute tests and evaluate the regression test selection technique on. The first application is an internal banking application used for producing a wide variety of different types of balance

\(^3\)http://depfind.sourceforge.net/
\(^4\)http://www.ietf.org/rfc/rfc1321.txt?number=1321
\(^5\)http://emma.sourceforge.net
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Log in</td>
<td>(A) Successful login and an empty customer page is displayed</td>
</tr>
<tr>
<td>2 Enter an incorrect customer no and click OK</td>
<td>(B) Error message</td>
</tr>
<tr>
<td>3 Enter a correct customer no and click OK</td>
<td>(C) Customer page is displayed</td>
</tr>
</tbody>
</table>

Figure 2. An example of 3 test cases in a test task contained in a scenario

statements for an account. The second application tested is a customer application for analyzing account transaction data used by corporate customers. This application is a part of another application, running in a different environment, where communication and synchronizations between the systems occur in run-time. The third application is an application used mainly by employees to manage electronic identities. This application communicates with special encryption hardware installed on a dedicated machine.

5.5 Test Cases

The test cases are defined as one scenario for each application. Each scenario contains a series of tasks that should be performed by the test engineer to verify the application’s functionality. Each task is made up of a number of test cases executed one after another. An example of a task is shown in Figure 2.

All test cases in a scenario were originally ordered in a single series even though there were test cases not conceptually dependent on each other and hence did not require to be executed in the specified order. For example, test case 3 in Figure 2 is not dependent on test case 2 to be completed before it can be executed. This is since it is not necessary to enter an incorrect customer number before a correct customer number may be entered. Since a login is always required before using the application, both test cases 2 and 3 are however dependent on test case 1 to be able to execute. To organize independent and dependent test cases and tasks, a directed graph was constructed where the edges represent test instructions and nodes represent expected results. Figure 3 shows the graph representation of the test scenario shown in Figure 2 extended with one more task. The three tasks (1,2), (1,3) and (1,4,5) can be executed independently from each other but the test cases within the test tasks must still be executed in the specified order.

5.6 Data Collection

We collected dependency data and coverage data for a baseline version of the system, in this paper denoted as V0, and collected MD5-signatures for two other versions of the system, denoted as V1 and V2 respectively. Version V0 vs. V1 is an application release (see Section 4), while V0 vs. V2 is an infrastructure release. The release V1 was the subsequent release to V0 while V2 was released prior to V0. Using a release prior to the baseline may be viewed as the organization discovers severe problems in the current release and chose to revert to a prior version of the system but only after retesting.

The procedure for the data collection and analysis was as follows:

1. Collect dependency data for V0
2. Execute the tests on V0 to collect class usage data for each test case using the coverage analysis tool
3. Generate MD5 signatures for V0, V1 and V2
4. Compare MD5 signatures to produce the list of changed classes between V0 and V1, and V0 and V2
5. Extract test cases that use the changed classes or use classes that are directly or transitively dependent on changed classes in V1 and V2.

We constructed a dependency graph for the baseline version and stored the data in a relational database. There were 1 459 079 dependencies between the classes in the system.

5.7 Analysis

Collecting dependency information and creating class firewalls for changed classes was time consuming and required significant computing resources for the large system studied. As an example, the extraction of the almost 1.5 million dependencies took more than two hours on a PC running Windows 2003 server having four Intel Xeon 2.4GHz processors and 2GB RAM.

An approach to reduce the analysis time and the size of the regression test suite further is to re-test only those test cases executing changed classes. By selecting only test cases indicated by the coverage analysis

6Not counting the core Java classes (java.*).
to execute changed classes, the dependency analysis and mapping between the class dependency diagram and the test case class usage can be avoided. Another motivation for using this approach is that only test cases that in the baseline version of the system actually executed a changed class are included in the test suite, leading to a higher precision. This approach is similar to the modification-based technique proposed in [18] and the modification-traversing approach described in [14]. However, a difference compared to those approaches is that a test case is selected if it executes a changed class even though the changed code in itself is not necessarily executed. We compare the results from using this more "light-weighted" approach to the results of applying the full class firewall regression test selection technique. In this paper we denote this approach change based selection.

It is indicated that the location of a change affects the number of test cases selected [3]. For example, Koju et al. report that more changes and changes to frequently used methods tend to select more test cases [6]. It was thus interesting to investigate to what extent this indication is valid for this large scale industrial system. The studied system uses services provided by systems hosted in mainframe computers within the bank. Communication with these systems is performed via specially developed application programming interfaces (APIs). A jar-file containing a set of Java classes implementing this API are used by the system to communicate with the mainframe systems. The services in the mainframe systems are however used by many other systems in the bank and since systems evolve these services also evolve. This leads to that the API classes are frequently changed. The API classes can be automatically generated from the specifications of the services and thus no manual work is needed to update them. This API is essential for all applications in the system and thus there are many dependencies to these classes. By removing the API classes from the list of modified classes before reducing the test suite we could get an indication of the degree of impact these API classes have on the test suite reduction.

6 Results

The results of the analysis are described in three parts below. First is the result of selecting individual test cases independent from each other. Then the results from the selection techniques when considering the test tasks are presented. Finally we present the results when test cases and test tasks are considered from a tester’s perspective, i.e. how the results affects the execution effort in practice.

6.1 Individual Test Cases

We conducted the class firewall test selection as well as change based test selection for both V0 vs. V1 (application release) and for V0 vs. V2 (infrastructure release), with and without the API classes. Figure 4 shows a histogram over the percentages of test cases selected using all four approaches. A1, A2 and A3 represents each of the three applications tested, and AVG is the average of the result for the three applications. FW=Firewall technique including all classes, CB=Change based technique including all classes, FW-API=Firewall technique with the API excluded, CB-API=Change based technique with the API excluded.

![Figure 4. Test suite reduction for individual test cases.](image-url)
in the system were included (67% of the test cases are selected, see AVG for FW in V0 vs. V1). However, as mentioned above, the dependency extraction alone took more than two hours to perform. Thus the data collection and analysis effort for using the class firewall regression test selection technique is expensive and the cost may not be saved by executing a 30% smaller test suite. For V0 vs. V2 the result was even lower, only 10% of the test suite could be reduced (90% AVG for FW, V0 vs. V2). The API classes did not have any large effect on the test suite when used together with class firewall selection technique. For two applications studied, no further reduction was indicated when excluding the API classes from the analysis (FW vs. FW-API for A2 and A3 in both V0 vs. V1 and V0 vs. V2), and the highest reduction was 18 percentage points (A1 in V0 vs. V2).

The histogram shows a considerable reduction of the test suite size for the application release V0 vs. V1 using the change based selection technique with the communication API excluded (AVG for CB-API, V0 vs. V1). The calculated average gives an 80% reduction of the test suite. For the infrastructure release V0 vs. V2 a 25% reduction was reached with the change based selection technique where the API classes were excluded (AVG for CB-API, V0 vs. V2). A possible reason for this smaller reduction is that V0 vs. V2 is an infrastructure release where changes were made mostly in core components used by many applications thus probably affecting many test cases.

6.2 Effect on Test Tasks

![Figure 5. Number of test tasks required to be executed](image-url)

A test case is included in a test task and a test task may contain several test cases. Thus if at least one selected test case is included in a test task, the test task must be executed. Figure 5 shows the percentage of the test tasks that need to be re-executed for the versions studied. This histogram indicates that using scenarios together with either the class firewall or the change based selection technique and including all classes in the system does not give any major reduction of the number of test tasks that must be executed (AVG for FW vs. CB in V0 vs. V1 and in V0 vs. V2). There are applications where all test tasks are selected and thus nothing is gained from using the selection techniques compared to using re-test all (FW for A1 in V0 vs. V1 and FW, CB for A1 and A2 in V0 vs. V2). However, a considerable reduction of the suite (more than 70%) was also reached for the first version using the change based selection technique when the communication API was excluded (AVG for CB-API in V0 vs. V1). However, for the V0 vs. V2 almost all test tasks still need to be re-executed, on average only 13% could be saved using this combination (AVG for CB-API in V0 vs. V2). This indicates that the type and location of changes have a large impact on the number of test cases selected using the techniques.

6.3 Practical Effects

![Figure 6. Test suite reduction including not selected but still required test cases](image-url)

Sometimes, an expensive activity in testing is the examination of the results from the test cases to check whether the actual results are equal to the expected results. An approach to further reduce costs is to view a test case, not selected but still required, only as a transport to get to the next dependent test case. With this view the examining of the actual results may be skipped and thus the cost of testing may be further reduced. Furthermore, depending on the nature of the
7 Threats to Validity

This section discusses the validity threats of the study. We focus on internal and external validity as described in [17]. Since the study was a technical evaluation performed without including users or test engineers or other personnel from the studied organization, no social or psychological validity threats are present.

7.1 Internal Validity

The internal validity is concerned with the study environment and the reliability of the result.

One threat against the validity of the study is the risk of have missed components in the analysis, thus affecting the results. For example, if a class file were corrupt and not analyzable, it may have been partly or fully excluded from the analysis. Since there were more than 50,000 classes in the two versions of the system, it was not possible to verify in detail all outputs from the tools. Instead, the output from the tools were logged and inspected briefly after execution.

Also, there is a risk that the scripts and commands used for search and filter of the data contain bugs and due to the large data sets the results could not practically be verified. However, most of the commands and scripts were first tried on a small controlled data set to verify their output before they were used on the larger data sets collected from the system. We also performed a “sanity check” to verify that the output from the scripts and commands were realistic.

7.2 External Validity

Threats to external validity concern the ability to generalize from the study to a larger population.

The possibilities to generalize from this study is limited since only a single system was studied. The system is however a real world industrial system of considerable size and thus the results are still considered relevant. The system is running on J2EE application servers and thus the system is implemented in conformance with a well known industrial standard and thus the system may be considered somewhat representative for similar types of businesses in other organizations.

8 Conclusions

We have presented an empirical study where the class firewall regression test suite reduction technique and a change based technique were applied on two versions of a large scale industrial system. The reduction techniques were evaluated using scenario based testing on system level and only Java byte code was used in the analysis.

Using the class firewall regression test selection technique together with scenario based testing, not all test cases were selected. The scenarios were divided into tasks where a task could contain one or more test cases required to be executed in a specified order. Since not all test cases selected by the reduction techniques were located in the beginning of their tasks, there were test cases that were required to be executed even though they were not selected for testing, leading to a larger test suite than produced by the techniques.

The coverage analysis identified all classes exercised by each of the test cases. This information was used for conducting change based selection, i.e. selecting those test cases exercising only modified classes in new versions of the system. The subset of the test suite selected using this approach was smaller compared to the subset selected by the class firewall selection technique.

As mentioned in [3] the locations of changes have impact on the size of the test suite selected. However, using the class firewall technique only a small difference in the size of the test suite could be noticed when excluding a set of core classes from the analysis for the first version analyzed. On the other hand, applying the
change based technique in combination with excluding these core classes resulted in an 80% reduction of the test suite.

Acknowledgements

We would like to acknowledge FöreningsSparbanken AB for sponsoring, and all personnel at FöreningsSparbanken AB for assisting in conducting this study. The work is partly funded by the Swedish Research Council under grant 622-2004-552 for a senior researcher position in software engineering.

References


