A Case Study on Regression Test Suite Maintenance in System Evolution

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

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

Abstract

Regression testing is important to validate existing functionality of a system after upgrades and may be conducted very efficiently by automation. However, when a system is maintained the automated test suites must also be maintained to keep the tests up to date with the system. Even though test suite maintenance can be very costly we have seen very few studies considering the actual efforts for maintenance of testware. In order to explore the question of test suite maintenance we conducted a case study on an evolving system with three updated versions changed with three different change strategies. Test suites for automated unit and functional tests were developed and executed for the baseline application, and used for regression testing the extended applications. The change strategy used for changing the baseline system had a high impact on the effort required to maintain the test suites. With one strategy more changes were made in the tests code than in the system that were tested and with another strategy no changes were needed for the unit tests to work.

1 Introduction

Most systems developed today are developed in an evolutionary fashion [17, 12]. An advantage of this approach is opportunities to reuse efforts invested in software development in many subsequent releases. Testing and test automation is an area that specifically benefits from the evolutionary approach. By enabling reuse of investments made in test programs and environments, also referred to as testware, higher costs for developing test automation support can be justified and hence higher quality software can be achieved.

Testing of existing functionality from an earlier release of a program is referred to as regression testing [11]. This is very important for users upgrading to the new release; the functions that used to work in the earlier release must work in the new release while the tolerance towards errors in new functions may be higher. When using automated tests, regression tests may be conducted very efficiently according to Fewster and Graham [6, page 9]. However, they also conclude that maintenance costs are more significant for automated testing than for manual [6, page 191]. Despite this, we have seen very few studies considering the actual efforts for maintenance of testware in an evolutionary environment.

Automated test suites are very sensitive to changes in the interfaces between the system under test and the testware. In the case of GUI testing, an added item in a scroll list, or even another test environment with a different screen size may destroy the automated tests. Further, there is a risk that even though the test suites are running, they become obsolete or redundant for the new release. The test cases may still be possible to run, but they do not fulfill the original test purpose any more. E.g. if some range limits have changed, two test cases testing values within and outside the range, may both test the within range situation, since the range is changed. Binder [3, pages 767-770] refers to a case study where 90% of the test cases became redundant.

In order to explore the question of test suite maintenance, we conducted a case study on an evolving system, the Drawlets application [15]. Starting with the original application, three different versions of an extended Drawlets application have been developed by other researchers [13]. Test suites for automated unit tests and functional tests for the original baseline system were used for regression testing the extended applications. On the unit level, test are run using JUnit
and on the functional level, the tests were conducted using Abbot [18]. In this paper, we report on experiences from the case study and the types of problems that arise during the regression tests. By performing the tests on three different versions of the changed system, we can analyze how different approaches to change interact with the regression testing.

Section 2 contains related work and the case study is described in Section 3. General observations are reported in Section 4 and details of the case study operations are presented in Section 5. Section 6 discusses the results and Section 7 gives directions for future work.

2 Related Work

In their extensive book on test automation, Fewster and Graham [6, chap. 7] provide an overview of issues related to how to build maintainable test. They provide good advice, probably based on practical experience, but very little empirical justification is provided.

When discussing test suite maintenance, Binder [3] defines four kinds of decay for test cases in the evolution of the software and its testware:

- Broken test cases cannot run. They do not interface with the software any more, e.g. due to changed method protocols.
- Obsolete test cases do not fulfill their test purpose. E.g. a test case testing the limits of a range becomes obsolete when the range is changed.
- Uncontrollable test cases cannot be run with a deterministic result. E.g. a test case which may be impacted on by the mouse cursor, in addition to the test script.
- Redundant test cases test the same issues one or more other test cases. Redundant test cases may appear e.g. when a test case becomes obsolete.

Binder proposes a systematic method for regression test suite maintenance [3, page 769] e.g. to remove redundant test cases based on coverage analysis. This approach is however neither further analyzed with respect to efficiency nor with respect to success at all.

Regression test suite reduction is another area of related work that intends to select a subset of regression test cases, without considering the automation. Rothermel et al has conducted several studies on methods to select regression test cases, see e.g. [7, 16]. However, these are all conducted on a very small scale, and no studies analyze the effect of scaling up.

The “test-first” approach to software development has appeared within the agile methods movement [10], specifically in eXtreme Programming (XP) [2, 1]. The principle is that test cases for automated unit test are written first, then the code for implementing the function is written. Thereby, it is ensured that the test cases always are up-to-date. In combination with refactoring, i.e. continuous change of the code and the testware, the test suites and the system are maintained at the same time. However, for organizations having legacy software and testware, a change towards the test-first approach must be taken in steps.

This overview of related work inspires to launching a case study to investigate in more detail the effects of software evolution on automatic regression test suites.

3 The Case Study

To run our case study a baseline system having updated versions available was selected. We executed existing unit tests, and created and executed functional tests for the baseline. Then we updated and executed the test suites on the updated versions of the system. The work of adapting the test suites to the updated systems was analysed.

3.1 The System

The case study was conducted on the “Drawlet” framework, a system that has been proposed as a testbed for software evolution case studies [5]. The Drawlet framework is a Java version of the original drawing program “HotDraw”, developed by Kent Beck and Ward Cunningham [15]. It is an extensible framework that may be used to add drawing functionality to a host application. The baseline version of the system used in this case study was the 2.0 version of the Drawlet framework developed with JDK 1.2. This package, downloadable from the Internet, includes the complete source code for the system as well as test code for unit testing with the JUnit testing framework [8]. It also includes host applications with graphical user interfaces provided as examples on how to use the framework. The host application named “SimplePanel” was used as the host application in our study, see Figure 1. This host application provides a canvas to draw figures on, a toolbar with buttons representing the figures that may be drawn, and a toolbar with buttons for cut/copy/paste operations.

3.2 Evolution

Rajlich and Gosavi [13] used the Drawlet framework in a case study on unanticipated incremental change.
The baseline system was extended with new functionality using different change strategies. The functionality that was added was intended to give users ownership to the figures they create, thus hindering other users to manipulate them. The new feature was implemented by modification of about 100 lines of code but had big impact on the overall concepts of the system. The same ownership feature was implemented three times, using three different change strategies, resulting in three different new versions of the system. These three change strategies were: Change propagation, Refactoring and Role splitting, see Figure 2.

For this case study the original system was viewed as the baseline and the other three versions as enhanced versions requiring regression testing with the (possibly adapted) baseline test suites before released as new versions of the product. This case study focuses on the regression part of the testing, thus new tests that should be developed to test the new functionality is outside the scope of this paper.

3.3 Testing

In the baseline package there are 19 source files giving 385 unit tests with 1369 assertions testing the classes in the system. A graphical user interface to JUnit is also included to aid in running and evaluating the tests.

When we executed the unit tests with the baseline system there were 6 assertions that did not pass. We decided to ignore these for our case study since we wanted to change the baseline as little as possible.

No functional tests were included in the baseline package and to our knowledge no such tests for the system are available elsewhere. Thus, specification based functional tests were derived from the specification of the Drawlet system found in [9]. The procedure for creating the functional test was as follows: First, from the functional specification we categorised the functions of the system into the following 6 major categories:

1. Draw - a figure may be drawn on the canvas
2. Select - a figure may be selected
3. Cut-copy-paste - interaction with the clipboard
4. Move - a figure may be moved
5. Resize - a figure may be resized
6. Connect - a figure may be connected to other figures

These 6 major categories were the starting point for creating functional test cases. The major categories were further analysed and divided into subcategories depending on how the functions could be used from the GUI of the host application. A total number of 141 functional test cases were created for the functions of the baseline.

The functional tests were organised in an order where the result of one test case could serve as a starting point for another. For example, a test case where...
Figure 3. Case Study Process

a box was drawn was followed by a test case where a box should be moved. This approach “enables a lot of comprehensive testing to be undertaken by a series of short test cases.” [6, page 197].

The functional test cases were realised and automated using a record/playback tool for Java applications named Abbot [18]. This tool supports record/playback of keyboard and mouse events and also supports capturing the graphical user interfaces for creating assertions. When a test case had been executed manually the tool was used to save a snapshot of the drawing canvas as an jpeg file. These images were then used to create assertions where the tool compared the current drawing canvas to the saved image.

3.4 Case Study Operation and Analysis Procedure

To execute the baseline unit tests with the new version of the system the baseline test code must first be built together with the changed system code. If this configuration is not built successfully then either the tests or the system must be changed. We assumed that the tests should be changed based on the guess that the three updated versions have already been thoroughly tried out in the case study they originate from.

The case study operation used is divided into 4 phases, so Figure 3:

1. The testing environment is configured and set up, a phase not included in the analysis. This phase includes the creation of functional tests and verifying the baseline system and the baseline tests.

2. The unit tests are adapted to the updated versions of the system and executed. Each system should first be built and executed without the test code to verify they could be compiled and started. Then the test source code included with the baseline package is added to the directory tree of the new system resulting in a new configuration that should be built. If this configuration can not be built successfully the tests are changed and the configuration rebuilt. This is repeated until the configuration can successfully be built.

3. The build consisting of both the new updated version of the system and the (possibly adapted) unit tests are executed using the JUnit testrunner. If the testrunner reported problems the tests are changed and rebuilt and the tests executed again. This is repeated until the tests pass.

4. The automated functional tests configuration is loaded into the Abbot tool and paths are updated to point to the new updated system. Then the replay of the earlier recorded test cases is started and the baseline functional tests executed on the updated version of the system. If the tool reports problems those problems are analysed and the functional tests is changed. This phase is repeated until the functional tests passed.

The study operation is repeated with each of the three updated systems, the Change propagation version was randomly chosen to be tested first. Our analysis focused on the changes that were made in order to build the new configurations and to execute the tests on the updated versions of the system. The task of adapting the tests to the updated systems were recorded by logging the steps of the work during its progress. This log was then analysed together with the different versions of the system and the test cases.

3.5 Threats to Validity

An empirical study should be reviewed with respect to its validity to ensure that it satisfies established standards for high quality research. Several topologies for validity threats exist [14]. For quantitative or fixed design research, the internal-conclusion-construct-external validity model is frequently used [19]. For qualitative or flexible design research, other issues have to be taken into consideration. We have found a model by Padgett, summarized by Robson [14], be useful in the current work.

The model analyses three kinds of threats to validity: reactivity, researcher bias and respondent bias. Reactivity refers to that object under investigation behaves differently due to the presence of the researcher. Researcher and respondent biases refer to that the researcher and the respondent bring their background
and personal viewpoints into the study, and thereby
colour the results. In this case study, the "respondent"
is a software system; hence the only type of threat to
be considered is the researcher bias.

The model presents six strategies to deal with the
threats to validity. For researcher bias threats, triangu-
lation, peer debriefing, member checking, negative case
analysis and audit trail are actions taken to reduce
threats. Prolonged involvement may reduce the reac-
tivity and respondent bias threats, but increases re-
searcher bias threats. Actions taken in this case study
to reduce threats are presented below.

Firstly, the audit trail strategy is adopted during the
case study, i.e. full record of activities during the study
is kept. The researcher in charge has used his engineer-
ing logbook to enable tracing of the events during the
case study. This is also a source for the negative case
analysis, which means that alternative explanations are
sought for during peer debriefing, where the second au-
thor has acted as a quality assurance person for the case
study. The triangulation strategy is applied by having
three versions of the same system and apply the case
study to those in a randomized order. Last, but not
least, the main strategy for reducing the threats of re-
searcher bias is by using the Drawlets system, which is
developed by others, and successfully used in an ear-
er case study [13]. The time of involvement increases
the threats of researcher bias. The researcher has in
this case worked with the system during 6 months part
time to evolve it. However, the versions used in the case
study are developed by others, and the researcher has
only worked with them in order to understand which
changes should be done in the interfaces towards the
test programs. Member checking, which means return-
ing empirical data to respondents is not applicable to
this case study, as it does not interact with any human
respondents.

In summary, established strategies to reduce threats
to validity are launched to make the study as valid
as possible. We consider the results in general being
independent of the researchers. Still it depends on the
certain case observed, but this is inherent in the case
study methodology.

4 General Observations

The following subsections describe our general ob-
servations of adapting the tests to the new versions of
the system. Sections 4.1, 4.2 and 4.3 discuss the unit
tests for each of the three versions and Section 4.4 the
functional tests.

<table>
<thead>
<tr>
<th></th>
<th>Compilation</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Files modified</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>Setup modified</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>Tests modified</td>
<td>51</td>
<td>8</td>
</tr>
<tr>
<td>LOC modified</td>
<td>82</td>
<td>42</td>
</tr>
</tbody>
</table>

Table 1. Numerical data regarding number of
changes made to the test code

4.1 Change propagation

We randomly choose the Change propagation ver-
sion of the system to test first. The baseline tests could
not be successfully built with this version of the system.
The Java compiler reported 246 compiling errors in the
test code and thus the tests needed to be adapted.

When these compilation errors were corrected the
unit tests were executed using the JUnit testrunner.
The first test run resulted in 1240 assertions, 10 er-
rors and 110 failures. Thus further changing of the
test code was required. Numerical data regarding the
changes is displayed in Table 1.

The baseline system consists of about 17800 lines of
code and the unit test code about 4800 lines. In [13] it
is reported that 91 lines of code (0.5%) was modified in
the baseline to create the Change propagation version
of the system. We modified 124 lines (2.5%) in the test
code to adapt the tests to these changes in the baseline.
Thus we modified approximately 1.4 test code lines per
modified system code line.

Details of the changes made to the test code for this
version is described in Section 5.

4.2 Refactoring

As for the Change propagation version, the baseline
tests could not be built with the Refactoring version of
the system due to e.g. changed method interfaces. It
was however possible to build and execute the system
together with the tests we had adapted to the Change
propagation version. The Refactoring system was cre-
ated by modifications of 95 source lines, i.e. approx-
imately the same number of lines changed as in the
Change propagation version.

1In JUnit terminology an assertion is a test that is checked
and passed, an error is the result of an unanticipated problem
during the execution of a test and a failure is a test that was
executed but did not pass.
4.3 Role splitting

The baseline tests and the Role splitting version of the system could directly be successfully built and executed without making any changes to either the test code or the system. It was also possible to build and execute this version together with the tests we had adapted to the Change propagation version without errors or failures. This was possible since the strategy used for changing the system was to create new methods instead of changing the existing ones when implementing the new ownership role. Hence, there were both methods that could be called directly from the baseline test code and new methods that could be called by the adapted test suite code. In [13] it is reported that 87 lines of code were changed to create this version. Thus the Role splitting strategy required the least number of lines to be changed both in the system and in the tests.

4.4 Functional Tests

The first thing that happens when each of the updated versions begin to execute is that a dialog box is displayed that must be completed by the user before the drawing user interface is displayed, see Figure 4. In this dialog box a session owner id and name should be entered followed by OK.

![Figure 4. The login dialog box](image)

This change in the updated version of the systems had implications on the functional regression tests. Since the program stops the execution and waits for the dialog box to be closed this also stops the execution of the tests. Thus we had to extend the functional test configuration to complete the dialog box to make the tests to run. Since the tests were executed one after another, without closing the application in between, the dialog box only needed to be handled once at the beginning of the execution of the functional tests. After making this extension regarding the dialog box, all functional tests could be executed without further changes to the tests. There were however faults in the updated systems due to imperfect evolution that led to regression test failures. Since the functional tests were dependent on each other, these faults needed to be fixed to be able to execute all of the functional tests.

![Drawets Panel example](image)

**Figure 5. The top toolbar in one of the updated systems**

In the new versions of the system the GUI had changed slightly, see Figure 5, compared to Figure 1. Two buttons had been added to the top toolbar in the GUI. The first button displays the session owner dialog box when pressed, giving the possibility to change session owner. The second button’s caption shows the strategy used for making the changes to the system. Since nothing happens when this button is pressed we believe this button was just used for identification in the previous case study which the updated versions originates from. These two new buttons did not cause any problems for the functional tests since our strategy for creating assertions included only the drawing canvas and not the toolbars.

We consider the effort required to make the changes to the functional tests as minor since we only needed to complete the first dialog box to make the regression tests to run.

5 The Case Study Operation in Details

This section and its subsections describe the details regarding how the baseline unit tests were adapted to the Change propagation version of the system. The Change propagation version was selected randomly but should be applicable to the Refactoring version also since the resulting test suites may be used on this version also.

5.1 Setting up the Environment

First, the baseline system was installed and executed to verify that it was runnable and usable. The unit tests included in the baseline package were then executed with the JUnit test runner to verify they worked properly with the baseline system. Then the functional tests were executed manually and recorded with the record/playback tool. The canvas was captured after executing each test case and assertions were created. The recorded functional tests were then replayed using the tool to verify they worked properly and that all assertions passed for the baseline.
The next step was to install the Change propagation version. This version used a Java applet host application instead of the standalone Java host application used in our functional tests. The same changes were made to the standalone host application as had been made to the applet version. The changes were 5 lines of code that were added in the host application. No other modifications were done to the updated versions.

5.2 Unit Tests

First, the updated version was built and executed without the test code to assure the system could be compiled and started. Then the test source code files included in the baseline package were added to the directory structure of the updated system. The following steps were to build the configuration, fix compilation errors, execute the tests and fix test errors and failures until the unit test could be executed with the updated version without problems.

5.2.1 Compiling

The first build of the test code together with the updated system resulted in 246 compiling errors that should be fixed.

The first compilation error was a cannot resolve symbol for a method named setBounds that was called from the test code. This was since in this new version of the system this method’s name had been changed to secureSetBounds and had been extended with a new parameter, sessionOwnerId, to be able to check which user is about to change a figure. To correct this error a new instance variable representing a dummy session owner was created in the test case class. Then the method call was changed to secureSetBounds with the dummy session owner variable passed as the extra argument. The method calls were not changed by search-and-replace of method names since there were method calls that should not be changed. Instead, we manually analysed and changed the method calls pointed out to us by the compiler.

There were several methods that had changed names to secure... and required a session owner as an argument, e.g. secureSetStyle, secureSetSize, secureMove. All these errors were fixed the same way, i.e. the call was changed to the new secure... counterpart and the sessionOwnerId variable was passed as an extra argument. Both setup code and test case code were changed to fix these compilation errors. This was the only kind of changes that was made to the test cases in order to get them through the compiler.

5.2.2 Executing the Tests

When all test cases passed compilation all unit tests were executed using the JUnit testrunner included in the baseline package. As mentioned in Section 4.1 the first test run resulted in 1240 assertions, 10 errors and 110 failures that needed to be fixed.

The first error was a Java NullPointerException since the baseline test code had not assigned any owner to the figure involved in the test. After assigning the dummy session owner as the owner of the figure before the figure was used this test passed. Similar problems were found in 35 different locations in 10 different test code files leading to both errors and failures. These changes resulted in 99 fewer failures and 5 fewer errors, leaving us with 5 errors and 5 failures (not counting the ignored 6 failures from the baseline).

Next problem resulting in failures was that in some circumstances the session owner must be set in the canvas in order for the updated system to work properly. During a normal execution of the updated system this is done as the first step before the drawing user interface is displayed by showing the dialog box where the user should enter session owner id and name and press OK, see Figure 4. To make the tests run, the test code must also set the session owner in the canvas class before calling methods requiring a correspondence between the session’s and the figure’s owners. However, the variable representing the session owner in the canvas class was declared as protected. Hence, it could not be set directly from the test code but must instead be set via some publicly accessible method in the canvas. The only method we could use to set the session owner object in the canvas was a public method named getNewID. When this method is called it creates an owner identity object and then calls a public method named getNewOwnerID on this newly created identity object. This getNewOwnerID method in the owner identity object displays the dialog box where the session owner should be entered, stopping further execution of the tests. Using this way of setting the owner the tester is required to supervise the execution of the tests and complete the dialog box manually in those places where a session owner is required by the system during the regression test run. This change was made in 5 locations leading to the dialog box must be completed 10 times during a test run.

Another change required to make the tests run was that some compound statements had to be split up due to how the new ownership concept was implemented. As an example, the compound statement

```
TextLabel tl = newTextLabel();
```

was changed to:

```
TextLabel tl = new TextLabel();
```
tl.setFigureOwner("MyName", sessionOwnerId);
tl.secureSetString("This is test text", sessionOwnerId);

This split up was required since the constructor in the class TextLabel taking a string as parameter calls the new secureSetString method to set the string. This method requires that a figure owner is set in the text label object in order to set the string, and since no owner is set by the constructor the check for matching session and figure owner fails and the string is not set. To get this to work we split up the compound statement and inserted a setFigureOwner call to set the figure owner before the string was set. Since this change uses the default constructor when creating the TextLabel object the string had to be set explicitly after the figure owner had been set using secureSetString.

After these changes all tests passed but those that did not for the baseline system.

6 Discussion

The changes made to the baseline Drawlet system to add the ownership features had implications on the tests. There were both internal implementation changes inside methods as well as changes to methods’ protocols, i.e. changed method names and extra parameters to methods. The protocol changes caused compilation of tests to fail since methods that were called in the original test code did not exist any longer. However, in one version, changed with the Role splitting strategy, methods were added instead of changed and thus the methods’ protocols remained the same and the tests could compile without errors. This led to that the Role splitting strategy was in this case study superior the others in terms of effort required to make the regression tests run. However, nothing can be said of the quality of the (adapted) baseline tests when running them on the new versions since no coverage analysis or other analysis of what the baseline tests actually test in the new versions were performed in this case study.

Changing the test code to call the new methods was not made by search and replace of method names since there were method calls that should not be changed. For example, there were calls to methods named setBounds on instances of the class Rectangle that is a part of the Java API that should not be changed. Instead, we let the compiler point out those locations in the code were the changes needed to be made and changed each one manually. This is an implication of the feature in object-orientation that several methods may have the same name in different classes, or even within the same class. This hinders the use of a naive search and replace.

The extension of the number of parameters to methods was fixed by creating a dummy session owner that were assigned a value of our choice. This seemed to work in this case study since the tests passed. However, nothing can be said about whether the test code requires more sophisticated changes than just passing the extra parameter in order to actually perform the same tests on the new implementation as it did in the baseline. Our belief is that a good knowledge of the changed implementation is required during test maintenance in order to not causing errors or failures and to actually make the test cases perform the same tests on the new implementation.

The internal implementation changes inside methods, e.g. adding the check for session owner to match the figure owner, led to problems when executing the unit test cases, both errors and failures. The adaptation of the tests to set the session owner in the canvas led to the tester must supervise the execution of the tests since the session owner dialog box must be completed for the tests to continue. Perhaps Java’s reflection could have been used to solve this but we believe it would probably require more effort to implement. A publicly accessible method in the canvas that could be used to set the owner identity would have made it easier to adapt the tests. If a “test-first” strategy had been used perhaps this problem had been discovered during the implementation and an implementation more suited for testing would be the result.

There were also tests cases that required calls to be split up and extended due to the changes in the implementation, e.g. the call to the TextLabel constructor. Using a “test-first” approach this would probably had been identified earlier and the constructor could have been implemented differently.

The only change to our functional tests was the addition of response to the first dialog box asking for the session owner. No other changes were needed to make the functional tests run. Thus the implementation of the ownership features had only minor affects on our functional test cases. Should however we had chosen to include the tool buttons and other graphical elements in the assertions, then all of our test cases would have failed since there were two new buttons added to the top toolbar in the new versions of the system. Thus, we agree with among others Fewster and Graham [6] and Burk and Coyner [4] that a tester should be very careful when the GUI is used in automated tests.

If test execution errors or failures are found in an industrial setting the assumption may be that the errors and failures are due to problems in the system under test. This assumption can turn out to be costly if it
shows after a long time that the failures were instead due to problems with the tests. In our case study we made the assumption that all errors and failures were due to problems with the test code. There were however some faults in the new implementations, and perhaps it took us longer to correct these due to our faulty assumption. Thus, during regression testing and test maintenance the best assumption should rather be that errors and failures are due to the system under test, the test code, or both. Further, the faults in the changed system caused some of the functional tests to fail. This hindered further execution of the tests since the tests were dependent on each other. This could be solved by not creating sequences of tests but instead explicitly set up the prerequisites for each test case independently and run them independently. However, in this case we only needed to add one prerequisite for our functional test, the completion of the dialog box, to make them all run. If all tests would have been set up independently, the completion of the dialog box would probably have been required to be added for each of the independent test cases and this would have required a greater effort.

7 Future Work

The Role splitting strategy seems to have positive effects on the efforts required to make changes to systems and regression test suites. Also, splitting the roles by splitting up a method into two similar methods that are used by different roles may have positive effects on the system. Since the baseline tests can be used with new versions of system to test the parts remaining from baseline it may be easier to determine what new tests must be developed for the new parts of the system. There may however be a drawback with developing new parts instead of changing existing parts. A new set of tests must perhaps be created to test the same features in the new parts that the baseline tests already test for the baseline parts, thus test code is duplicated. Also, perhaps more effort is required to create new tests compared to the effort required to adapt existing tests to their changed implementation. A direction for future work is to launch a new case study to investigate this matter. It would also be interesting to investigate the effects of using the Role splitting strategy for several subsequent releases of a system.

We believe that some of the problems found would have been discovered earlier if a test first approach had been used, e.g. the compound statement split-up and the requirement to complete a dialog box during the execution of the tests. We are preparing a study where one set of subjects add the ownership feature to the Drawlet system using a “test-first” approach and another group using a “test-last” approach. This study is aimed at investigating differenced between the solutions resulting from the two approaches.

This case study was aimed at investigating the effort required to adapt baseline tests to make them compile and execute together with new versions of a system. An interesting follow-up study would be to investigate the quality of the resulting test suites. Interesting issues would be to analyse whether the adapted tests actually tests the same features in the updated system as they did in the baseline.

Acknowledgements

The authors would like to thank Václav Rajlich and Prashant Gosavi for giving us access to the three updated Drawlet versions we used in this case study.

References


