Introducing Software Test Automation and Test-Driven Development: 
An Experience Report

Lars-Ola Damm\textsuperscript{1, 2} \hfill Lars Lundberg\textsuperscript{1} \hfill David Olsson\textsuperscript{2}

Lars-Ola.Damm@Ericsson.com \hfill Lars.Lundberg@bth.se \hfill David.Olsson@Ericsson.com

Abstract
This paper identifies and presents an approach to software component-level testing that in a cost effective way can move defect detection earlier in the development process. The approach includes a test automation tool using a combination of framework driven and data-driven script techniques. A department at Ericsson AB introduced the tool for component-level testing in two projects together with the concept test-driven development (TDD), a practice where the test code is written before the product code. The implemented approach differs from how TDD is used in Extreme Programming (XP) in that the tests are written for components exchanging XMLs instead of writing tests for every method in every class. This paper describes the implemented test automation tool, how test-driven development was implemented with the tool, and experiences from the implementation. Preliminary results indicate that the concept decreases the development lead-time significantly.

Keywords
Software test automation, test-driven development, component testing, unit testing, test tool.

1. Introduction

Studies indicate that testing accounts for at least 50\% of the total development time [12, 13]. One reason for this is that the verification activities late in development projects tend to be loaded with defects that could have been prevented or at least removed earlier (when they are cheaper to find and remove \cite{6, 12, 23}). When many defects remain to be found late in a project, schedules are delayed and the verification lead-time increases \cite{15}.

A software development department at Ericsson AB (from now on referred to as the department) develops component-based software for the mobile network. The department wanted to decrease the verification lead-time and avoid the risk for delayed deliveries by introducing a new tool and process for automated testing on a component level. To achieve this, they needed to determine what was required of the tool, process and organization.

Thus, the paper has four main questions to answer:
• What characteristics should a test automation tool for component level testing have?
• What is an appropriate process supporting the use of such a test automation tool?
• What aspects need to be considered when introducing a new process and tool for component level testing?
• What is the expected lead-time difference for the projects that introduce such a tool and process?

The method used for answering the first two questions was a combination of analysis of lessons learned from previous improvement attempts together with a thesis study \cite{8} on how to increase the test efficiency at the department. The thesis study included qualitative and quantitative enquires, analysis of project statistics, and a literature study. Answers to the other two questions were captured from qualitative and quantitative interviews with users of the introduced concept.

The paper is outlined as follows. A background is given in Section 2 and Section 3 presents the choice of process for the concept, i.e. test-driven development. Section 4 presents the actual implementation together with some observations, lessons learned and expected lead-time gains. Section 5 provides a discussion where the new test automation tool is mapped against related techniques for test automation and then it describes what is required of an organization and its products before being able to implement such a concept. Sections 6 summarizes the work through some conclusions.

\textsuperscript{1} Dept. of Software Engineering and Computer Science, Blekinge Institute of Technology, Box 520, SE-372 25 Ronneby, Sweden.

\textsuperscript{2} Ericsson AB, Soft Center, SE-372 25 Ronneby, Sweden.
2. Background

The approach used before the introduction of this new concept comprised a test tool (called DailyTest) that could test isolated components before delivery to the function test department. In this context, a component is an executable asset that communicates with other components through common interfaces. Further, a component contains about 5-30 classes and the products that the department develops consist of about 10-30 components each.

DailyTest could execute scripts consisting of simple commands with references to files containing XML requests; for example, the script in Figure 1 loads a component and then sends a sequence of requests on it (the requests are specified in ‘test.files’).

```
CONFIGURATION NAME TestComponent1
CONFIGURATION CREATE_REQUEST ComponentCreate.xml
TEST DELAY 10
TEST_CONN DATA ADD test.files
TEST_CONN NAME TestComponent1
```

**Figure 1. DailyTest - script example**

The reasons why the tool did not work satisfactorily were that it was limited (few commands and no looping) and more importantly, it was not properly integrated with the development process. When the department had realized this, the earlier mentioned thesis study [8] evaluated the test process with the purpose to identify tool and process changes that would increase the test efficiency. The thesis study showed that the cost for finding and fixing defects increases significantly the later in the development process they are found. This is also considered a common fact in software development [6, 12, 23]. Since the thesis study also discovered that the developers normally put little effort on testing isolated components, the thesis suggested that this is where to focus the improvement efforts.

Testing of isolated software components is the first test level in the department’s quality assurance strategy, Basic Test in Ericsson terminology. As in ordinary unit testing, the purpose of Basic Test is to verify the design specification. However, as opposed to unit testing, Basic Test is not a white-box test technique since the components’ interfaces hide the internal component design. Nevertheless, since it is the product developers that perform Basic Test on the components, they still know the internal design structure. This kind of testing is sometimes called gray-box testing and it is considered powerful because when you know how the code looks on the inside, you can test it better from the outside [19]. The reason why the developers start doing testing on a component level is because it is not cost-effective for them to test every class/method. The reason for this is that the departments’ products have a higher testability on the component level, i.e. the components are independent executables and the XML requests that the components send between each other are easier to verify (testability is further discussed in Section 5.2.2).

After determining that the improvement efforts should focus on the Basic Test level, the thesis study determined that the main reasons why the developers at the department did not Basic Test all functionality was because of insufficient test tools (e.g. DailyTest) and process deviations when the deadline pressure was high due to delayed schedules. When the development activities were delayed, the projects tended to deliver the code untested hoping that it in a miraculous way would work anyway. Likewise, this phenomenon seems far from uncommon in the software industry [16, 21].

From the experiences and findings discussed above, the thesis study suggested a new tool for how to automatically Basic Test the products at the department, which they thereafter implemented and introduced in two upcoming projects.

3. Test-driven development

To make sure that the new Basic Test tool would not become a shelfware, the department put considerable efforts in integrating the tool with the development process and they chose between keeping their previous standard process or to introduce the new concept test-driven development (TDD) [3].

The main difference between TDD and a typical test process is that in TDD, the developers write the tests before the code. A result of this is that the test cases drive the design of the product since it is the test cases that decide what is required of each unit [2, 3]. Therefore, TDD is not really a test technique [2, 7]; it should preferably be considered a design technique. Furthermore, TDD simplifies the design and makes sure that the implementation scope is explicit, i.e. it removes the desire for gold plating since that will not make any more test cases pass [2]. Nevertheless, the most obvious advantage of TDD is the same as for test automation in general, i.e. the possibility to do continuous quality assurance of the code (including regression testing) [10]. This gives both instant feedbacks to the developers about the state of their code and most likely, a significantly less percentage of defects left to be found in later testing and at customer sites [21]. Further, with early quality assurance, a common problem with test automation is avoided; that is, when an organization introduces automated testing late in the development cycle, it becomes a catch for all defects just before delivery to the customer. The corrections of found defects lead to a spiral of testing and re-testing which delays the delivery of the product [20].

The most negative aspect concerning TDD is that in worst case, the test cases duplicate the amount of code to
write and maintain. However, this is the same problem as for all kinds of test automation [14], and to what extent the amount of code increases depends on the granularity of the test cases and what module level the test cases encapsulates, e.g. class level or component level. Thereby, since the department would use TDD on a component level, they would decrease the amount of test case code to write and maintain. Additionally, in comparison to classes/methods, it is easier to automate uniform interfaces that use XML as data format and that are more robust to changes.

Nevertheless, the department foremost chose to use TDD because it can eliminate the previously mentioned risk for improperly conducted Basic Test. The reason for this is that when the test cases are developed before the code, it is consequently impossible to deliver the code without developing the test cases. Meanwhile, there is no reason not to Basic Test the product when the executable test cases already are developed.

Finally, the purpose of introducing the new concept was not to replace manual testing with automated testing, especially since the developers only performed ad hoc Basic Test before. Instead, the department wanted to increase the amount of tested code in Basic Test. Larger benefits with test automation come not only from repeating tests automatically, but also from executing tests that were never executed at all before [14, 23].

4. Description of the Basic Test concept

After giving some background information on how the tool and process were selected, Sections 4.1 and 4.2 give a technical tool description. Section 4.3 describes how the tool was integrated with the development process at the department. After that, Section 4.4 lists some observations and lessons learned and finally, Section 4.5 presents the expected lead-time gains from introducing the concept.

4.1. Choice of tool and language

Since the purpose of Basic Test is to test the components in isolation, the Basic Test tool needed to be attached to the components’ interfaces, i.e. simulating the surrounding components. This attachment is demonstrated in Figure 2.

DailyTest, the previous Basic Test tool the department used, attached to the components in a similar way as in Figure 2. The reason why it was not preferable to enhance DailyTest instead of developing a new tool was that to make DailyTest as powerful as needed, it would almost become a program language. Naturally, it is not beneficial to do that when there already exist several powerful standard languages such as C++ and Java. In fact, using an in-house developed language has the same effect as when using proprietary languages (i.e. vendor scripts), they make coding as hard as writing English without being able to use the letter N [19].

![Figure 2. Basic Test tool - attachment to components](image-url)
4.2. Test case syntax and output style

After deciding to develop an in-house Basic Test tool with C++ as test case language, the department chose to use the framework-driven approach when designing the tool, i.e. the tool isolates the component to test from the test cases through wrappers and utility functions (see Section 5.1.4). Figure 3 shows an example of what the test case syntax could look like in the new Basic Test tool. Note that the code in the figure is an extract from a method in a C++ program that has instantiated necessary objects; for example, the object ‘ToolSender’ has previously been initialized with arguments such as component name and port to send the requests to.

```c++
Tool.startTest("Test1");
theComponent.startComponent();
Tool.startTest("Test1:1");
ToolSender.sendMessage(Request.xml, ExpectedResult1.xml);
Tool.endTest();
Tool.startTest("Test1:2");
ToolReceiver.receiveMessage(ExpectedResult3.xml);
Tool.endTest();
theComponent.stopComponent();
Tool.endTest();
```

Figure 3. Basic Test tool - test syntax example

As illustrated in the figure, the tool provides startTest and endTest constructs that recursively can encapsulate all tests in groups of tests and requests. In addition, the tool contains several commands for handling component states and for sending and receiving requests. All actions are controlled by the tool and during the execution, the tool compares each sent/received request with its expected result and then logs the result in an XML file.

```xml
<Test name="Test1" status="Failed">
  <Statistics>
    <StartTime>2003:01:01-12.24</StartTime>
    <EndTime>2003:01:01-12.25</EndTime>
    <NumberExecutedTests>2</NumberExecutedTests>
  </Statistics>
  <Test name="Test1:1" status="Failed">
    <Request1>
      <Result>Ok</Result>
    </Request1>
    <Request2>
      <Result>Not Ok</Result>
    </Request2>
    <ResultFile>ComparisonResult.xml</ResultFile>
  </Test>
  <Test name="Test1:2" status="OK">
    <Request1>
      <Result>Ok</Result>
    </Request1>
    <Test>
    </Test>
  </Test>
</Test>
```

Figure 4. Basic Test tool – XML log example

Figure 4 shows an example of a log output from a test execution. The department chose XML as output format because it is already used as standard data format in their products and because nowadays, XML is a standard format to which many other tools and parsers easily can be attached. For example, it is easy to develop a GUI that parses the XML data into tree structures of test results (according to the tag structure in the XML). In such a GUI, the developers can monitor their test executions and the managers monitor the test progress.

Figure 5 displays the contents of the comparison result XML file that was specified in Figure 4 (for the test that failed). As can be seen in Figure 5, the XML file shows where the result XML file differed in comparison to what was expected.

```xml
<Request>
  <Assistance>
    <WantedSat>
      <Error>
        <ExpectedResult>
          <Id>59</Id>
        </ExpectedResult>
        <Received>
          <Id>60</Id>
        </Received>
      </Error>
    </WantedSat>
  </Assistance>
</Request>
```

Figure 5. Basic Test tool – ComparisonResult.xml

Since the developers should write the test cases in C++, it is also possible to divide the tests in different sub-methods. For example, the department chose to use this possibility to divide the tests into functional and non-functional (e.g. performance and robustness) tests that could be executed separately, determined by run-time arguments. Since it is hard to determine if performance requirements are met on a component level, the primary purpose of such tests were to serve as benchmarks, i.e. to monitor if the performance is getting better or worse as the product evolves [19]. However, the department sees a large potential in identifying performance and robustness problems earlier in the development process since those problems would be easier to manage at that level, e.g. finding a severe problem close to customer delivery is hard to manage without facing delayed deliveries.

4.3. Adjustments to the development process

Just providing a good tool does not ensure successful test automation; a tool only becomes as good as the people using it [23]. Since this tool was to be used with TDD (see Section 3), the test cases should be written before the code. As earlier mentioned, the department introduced TDD on the component level instead of the class level, i.e. the developers construct a set of test cases for each component instead of a set of test cases for each class. The test strategy was to capture all inputs and outputs of each component. Thereby, the tests represent...
the external design of each component. This also led to
test cases that could serve as a part of the design
documentation, replacing some of the old design (e.g.
component specifications). The result of this was a more
thorough design (in comparison to plain English, C++
does not leave room for misinterpretations) and that some
design time could be saved when being able to remove
some of the old design documentation. In fact, one
developer that started using the new concept stated that
the time required for constructing the test cases would be
earned back during coding even if he would never execute them (i.e. the gained knowledge during the test
case construction made it easier).

Figure 6 displays how the Basic Test concept was
incorporated with the process levels at the department.
The new activity “Basic Test design” is when the
developers construct the components’ test cases and it
comprises both implementation and inspection of the test
cases.

![Figure 6. Basic Test process at the department](image)

With a tool and a process for how to use the tool
established, there is only one major thing left to put in
place: a standard for how to write test cases. Otherwise,
you will end up with spaghetti tests that are hard to
understand and maintain leading to that the benefits with
test automation will be lost [10]. Test automation is
development and the test cases are software programs
testing other software programs; therefore, the test cases
are subject to the same design and construction rules as
the programs to test [23, 24]. Thus, the tests should
follow ordinary guidelines for structured programming so
that they become simple, reusable and easy to understand
and maintain.

Further, the tests should be independent [19] since this
enables the possibility to only execute individual tests
when short execution time is crucial. Additionally, when
having independent tests, the source of defects can only
be within that test, which makes it is easier to locate the
source of a defect. Finally, to enable daily regression
testing, the tests must be constructed for repeatability, i.e.
it must be possible to execute them without human
interaction [21]. To be able to track and repair the defects
found in regression testing, good naming conventions for
the tests are necessary. As can be seen in the XML log
example in Figure 4, all tests have a nametag. The
department used this tag to specify what feature each test
is supposed to verify. This also makes it possible to at all
times know the progress of each feature (percent
developed/passed).

To ensure that the developers would follow the
standard for writing test cases, someone should inspect all
test cases before letting the developers start implementing
the product code. Moreover, since developers in general
are used to structured programming they are good at
writing structured test cases [23], but far from all
understand testing [19]. The inspections must ensure that
the tests not just show that the code will work, but also
the opposite [25].

Another challenge the department had to address was
to implement the new Basic Test concept on products that
already existed since several years and therefore, the
components contained a lot of old functionality that did
not have such tests. Since developing tests for all old
functionality directly would be far too costly for a single
project to handle, the department chose only to develop
tests for new and modified functionality. Henceforth, the
strategy was to develop tests for more and more of the old
functionality during upcoming projects (i.e. in future
releases of the product).

4.4. Observations and lessons learned

During the implementation and introduction of the
new concept, several observations were made and some
lessons were learned. This section provides a list of those
that were identified from qualitative interviews with
managers and developers at the department after the
introduction of the concept.

The department established that…

…test automation requires high product testability.
According to related work, it is the product interfaces that
determine the opportunities for test automation [19]. The
importance of testability is further discussed in Section
5.2.2.

…it is important with a thorough framework design
because it makes test case development easier and is
robust to future changes. A similar experience is reported
in [23].

…test automation on a component level requires
adjustments to the product architecture since component-
level test architectures must have a tight integration with
their component architectures (this requirement is also
acknowledged in [22]).

…making people add new tests every time a new
defect is found requires continuous remainders and
monitoring. Otherwise, developers tend to deliver the bug
fixes untested. Furthermore, the authors in [21] state that
adding new tests for each fault strengthens the test suite.

…test execution speed is important. Other studies
support this experience with the claim that the faster the
test execution speed is, the more likely developers will execute the tests themselves [21].

...what gets measured gets done [15, 25]. When the department started the introduction of the new concept for Basic Test, the target projects gave it modest attention. However, when the projects started measuring the progress for number of test cases developed and passed, the usage rates increased significantly.

...beware of attempts to deviate from the agreed process. When in time-pressure, projects tend to neglect some activities which might give near-time benefits but that might be devastating in the long run [15]. Therefore, such deviations should not be allowed without being agreed on by all involved parties (e.g. the line organization).

...test-driven development makes people think through the design more instead of rushing to coding directly without knowing exactly what to implement yet.

...when introducing new ways of working, it requires significant efforts to convince people in the organization that the new methods will improve the productivity. If not succeeding to do this, the introduction of the new methods will most likely fail due to resistance among managers and developers to accept the new ways of working (further discussed in Sections 5.2.1 and 5.2.2).

...test tools easily become the excuse for every problem [9]. Reasons for a problem can be in either the test tool, the development environment or in the application under test; still, the Basic Test tool became the scapegoat for most problems since it is in the tool the problems first are discovered no matter where they origin.

...automated testing requires dedicated resources. As also considered a common fact [19, 23], test automation cannot be managed as a spare-time activity. In comparison, developing or buying the tool is rather cheap; it is introduction costs such as setting standards for test case writing, teaching users how to write good test cases, and tool maintenance that are costly.

...benefits from test automation are hard to obtain in the first project release. Other reports support this experience by claiming that upfront costs eliminate most benefits in the first project and benefits from regression testing are usually not realized until the second release [18].

...test automation should be introduced in small steps, e.g. as a pilot project to avoid taking unnecessary risks (if something is introduced in the wrong way but only in small scale, the cost of fixing the problem is most likely lower). This advice is also given in the research literature [10, 14, 19].

...minimizing maintenance costs is the most difficult challenge in test automation. The test cases must be robust to changes during bug fixes and in new product versions. Since this is hard to achieve, the most common problem in test automation is probably uncontrolled maintenance costs [19].

4.5. Expected lead-time gains

This paper does not provide actual results on costs and benefits of the introduced concept. However, preliminary project evaluations indicate significantly decreased fault rates. Further, Figure 7 presents the result of a study where all the developers that used the concept in product version X estimated their expectations in a questionnaire. On average, they estimated that the project lead-time would decrease more and more when using the concept (e.g. 25% in the third project). Also note that the developers did not think that the introduction costs in the first version delayed the project, i.e. they estimated gains already from the beginning.

<table>
<thead>
<tr>
<th>Product version</th>
<th>Expected lead-time difference (development time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version X</td>
<td>-2%</td>
</tr>
<tr>
<td>Version X+1</td>
<td>-19%</td>
</tr>
<tr>
<td>Version X+2</td>
<td>-25%</td>
</tr>
</tbody>
</table>

Figure 7. Expected lead-time gains with new concept

Another finding (from qualitative interviews with the project managers) was that not only decreased lead-time was the reason for using the concept; they thought that increased progress control (percent test cases executed/completed), and increased delivery precision (due to increased progress control and improved quality assurance) were at least as important.

5. Discussion

5.1. Related techniques for test automation

The purpose of this section is to relate the techniques used by the department’s test automation tool with other techniques used in the software test-automation industry. The presented techniques are described together with some of their pros and cons followed by a description of how the department’s tool relate to the presented techniques. However, note that the primary objective of this section is to benchmark the tool, not to perform a technique evaluation.

5.1.1. Capture-replay

The basic concept of capture-replay is that a tool records actions that testers have performed manually, e.g. mouse clicks and other GUI events. The tool can then later re-execute the sequence of recorded events automatically. Capture-replay tools are simple to use [4], but according to several experiences not a good approach.
to test automation, since the recorded test cases easily become very hard to maintain [10, 18, 19, 23]. The main reason is that they are too tightly tied to details of user interfaces and configurations, e.g. one change in the user interface might require re-recording of 100 test scripts [23]. Using capture-replay tools are like embedding constants in the software; every time the value of the constant change, the value has to be modified at all occurrences [22]. Further, it is considered doubtful whether capture-replay really can be classified as test automation because although the test execution is automated, the testers still have to verify the execution results manually [10].

5.1.2. Script techniques

A common approach to test automation is to write some kind of test scripts for the test cases. Script techniques provide a language for creating test cases and an environment for executing them [22]. Approaches to scripting varies in several dimensions:

- Simple scripts \textit{versus} highly structured scripts:
  Scripts may vary in complexity from simple linear scripts to scripts that are keyword driven with conditional and looping functionality and further to real programming languages [10].

- Standard scripts/languages \textit{versus} vendor scripts:
  Scripting tools either use a standard script programming language (e.g. Visual Basic, C++), or their own proprietary language (vendor script) [19].

- Scripts control flow and actions \textit{versus} input data controls flow and actions (data-driven testing):
  This choice is about whether the script or the input data controls the execution. “Data-driven testing is testing where data contained in an input test data file control the flow and actions performed by the automated test script” [23].

- Standard scripting \textit{versus} framework driven testing:
  Instead of operating against the product interfaces directly, framework driven testing adds another layer of functionality to the test tool where the idea is to isolate the software from the test scripts. A framework provides a shared function library that becomes basic commands in the tool’s language [18, 23].

The main advantage with script techniques is that they can be as powerful as ordinary programming languages. At the same time this is their major disadvantage since there is more that can go wrong, and more to update and manage [10]. The script technique to choose should be context dependent [19], i.e. the skills of the people together with the architecture of the product should determine which technique to choose.

5.1.3. Test-case generators

The most advanced technique for test automation is test-case generation. Boris Beizer has identified a few variants of test case generators [4]:

- \textbf{Structural test generators}: Generate test cases from the structure of the code. The problem with such generators is that they in best case can provide a set of test cases that show that the program works as it is implemented (design defects do not show) [4].

- \textbf{Data-flow-generators}: This variant uses the data-flow between software modules as base for the test case generation; for example, generation of XML files to use as input data in the test cases.

- \textbf{Functional generators}: It is difficult to generate code from functional generators because they require formal specifications that they can interpret [4]. However, when working, they provide a more relevant test harness for the functionality of the system since they test what the system should do (as opposed to structural test generators).

Additionally, some generators provide functionality to easily generate random test data. However, when generating random test data, it is difficult to predict the desired outputs. Consequently, when the execution cannot be verified automatically, it leaves extra manual verification work for the testers [4].

5.1.4. Technique comparison

As stated in Section 5.1.1, standard capture-replay tools are easy to use but the hardest to maintain. A good test case generator can save time during test design since it can generate several test cases fast and it has also the possibility to re-generate test cases in maintenance. However, when expected results need to be added manually or when the generator put extra requirements on the design documentation both development and maintenance costs will rise. Further, the generated test cases/executions of the test generators must also be checked manually to verify that they test the right functionality. Additionally, test case generators that provide and maintain test cases at a low cost might still not be cost-effective for the organization because the test cases they generate test the wrong things and therefore, they are not good at detecting defects.

Frameworks give the possibility to use wrappers and utility functions that can encapsulate commonly used functionality. Such mechanisms make maintenance easier because with well-defined wrappers and utility functions, an interface change only reflects the framework code instead of reflecting several test cases [22]. Framework development and maintenance require dedicated recourses; however, such efforts can repeatedly pay for themselves since the quantity of test case code to write
and maintain significantly decreases through the wrappers and utility functions [18].

Data-driven testing is considered efficient since testers easily can run several test variants and because the data can be designed in parallel with the script code [23]. This decreases both development and maintenance costs; in fact, some people state that the key to efficient testing is effectively created and utilized test data [19].

In practice, it is common that test automation tools use a combination of techniques; for example, data-driven testing can be combined with most other techniques, at least all of those presented in this paper.

5.1.5. Techniques used by the Basic Test tool

The Basic Test tool uses a framework-driven script technique for test automation. However, since the XML requests that are specified with the test cases contain some control data, the Basic Test tool is to some extent also data-driven. Further, the department has plans on making the test cases more data-driven by adding tool support for enabling more control logic in the XMs. Additionally, previous experiences in combining framework-driven and data-driven techniques seem successful, that is, the combination reaps the benefits of both data-driven and framework driven approaches because it leads to effective and maintainable test suites [23].

In order to even more decrease the costs for development and maintenance of test cases, future improvements could also include a limited variant of test generation; for example, generation of XML files from their DTDs (Data Type Definition).

5.2. Other considerations

Before implementing a new concept as the one described in this paper, both the organization and its products must fulfill some prerequisites; otherwise, the risk for failure is substantial. This section describes a number of such prerequisites in relation to the situation at the department.

5.2.1. Maturity of the organization

First, the development process that the developers follow needs to be mature enough. If the process is poor, test automation will not help [10, 19]. Preferably, the test automation effort should be easy to adapt to current practices; if the change is too great, the risk for resistance among the developers increases [17]. Furthermore, if the developers do not want to work as directed, they will not [15].

Thus, to change the behavior of the developers, they must be convinced that the new methods will help them and the business [15]; especially since many developers think that test script development interfere with their application work (in particular during time-pressure). At the department, the developers were aware of that neglecting Basic Test results in increased verification lead-time [8]. Therefore, they were open to improvements in Basic Test.

Another group of people that must be committed to the new methods are the managers because it is those that have to take the decision to accept the upfront costs with introducing test automation that most likely will impact short-term budgets and deadlines negatively [5]. Test automation is a significant effort that requires skills in programming, testing, and project management; test tools are just the tip of the iceberg [10, 19].

5.2.2. Maturity of the products

When introducing test automation, it is important that the software is testable, e.g. by providing interfaces that are easy to develop test cases for, robust to changes, and whose data is easy to represent in test cases together with expected outputs that the test execution tool automatically can verify against the received outputs. Further, a product with good testability contains built-in logging functionality for finding sources of errors found during testing [19]. The more testable the software is, the less effort developers and testers need to locate the defects [22]. Actually, testability might even be a better investment than test automation [19].

The department was aware of the benefits with having good testability long before they implemented the new Basic Test concept. As earlier mentioned, the department develops components that nowadays communicate through a common socket interface exchanging XML data. This architecture is the result of testability improvements that when implementing tools for test automation significantly reduces development and maintenance costs. Since the GUI part of the application is communicating through a common interface on top of the components containing the function logic, it is also possible to test the GUI logic on the component level. This is a significant advantage since GUI’s otherwise are hard to automate. In fact, almost any interface is easier to automate than a GUI; tying automated tests to specifics of the user interface is a maintenance trap [19]. Further, experiences show that the availability of programming interfaces for testing have a strong correlation to the development of powerful automated test suites [19]. At the department, a few hard tested interfaces still remain, but since they constitute a minor part of the execution flow, they were excluded from the initial test automation effort. Besides, this is in accordance with the common notion not to mandate 100% test automation [10, 19, 23].
6. Conclusions

The Ericsson department introduced a new test automation tool incorporated with an alternate approach to Test-Driven Development (TDD), i.e. TDD on a component level where the interfaces comprise socket connections exchanging XML data instead of classes and methods. With such an approach to test-driven development, robust and uniform component interfaces make test automation easier.

The test automation tool was developed as a combination of framework-driven and data-driven techniques because they minimize development and maintenance costs (primarily through encapsulation and reuse). As script language for the test cases, the tool use C++, the same standard programming language as the developers write the product code in, because:
- The developers are already familiar with it.
- It is more powerful than a script language (e.g. when the test framework lacks functionality for testing a feature it is possible for the test case developers to develop the support themselves).
- The developers can take advantage of programming tools already available, e.g. compilers and debuggers.

Additionally, the tool provides log results from the test executions in a flexible and portable XML format.

The software development department introduced the concept TDD to support the tool mostly because:
- When writing the test cases before the code, testing really happens.
- The test cases drive the design and make it more straightforward, i.e. with an explicit scope and no gold plating.
- TDD moves fault detection earlier in the development process when the faults are cheaper to find.

Altogether, our approach (which is based on TDD) makes testing a natural part of the development process; “if testing is applied only as a ‘necessary evil’ after coding, it is equivalent to manufacturing processes of 100 years ago” [11].

During the implementation of the new tool and process, some observations were made. In short, it is important that the components to test have high testability and that the test automation tool is thoroughly designed, tightly integrated with the product architecture, and provides adequate test execution speed. Further, test automation requires dedicated resources, involves significant upfront costs, and is hard to introduce successfully unless the organization and its processes are mature enough for it. Finally, maintenance of test cases is a difficult but also a very important challenge to master before achieving successful test automation.

Preliminary project evaluations indicate significantly decreased fault rates from the introduction of the new concept. Further, the developers that have used the concept have estimated that the project lead-time will decrease more and more for each new project version that uses it (see Section 4.5).

7. Acknowledgements

This work was jointly funded by Ericsson AB and The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.ipd.bth.se/besq).

8. References


