Observations and Lessons Learned from Automated Testing

Stefan Berner, Roland Weber, and Rudolf K. Keller*

Zühlke Engineering AG
Zürich-Schlieren
Switzerland

{sbn, row, ruk}@zuehlke.com

ABSTRACT

This report addresses some of our observations made in a dozen of projects in the area of software testing, and more specifically, in automated testing. It documents, analyzes and consolidates what we consider to be of interest to the community. The major findings can be summarized in a number of lessons learned, covering test strategy, testability, daily integration, and best practices.

The report starts with a brief description of five sample projects. Then, we discuss our observations and experiences and illustrate them with the sample projects. The report concludes with a synopsis of these experiences and with suggestions for future test automation endeavors.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Management – *Productivity, Software Quality Assurance (SQA).*

General Terms

Management, Design, Economics, Reliability, Experimentation.

Keywords

Software Test, Automated Testing, Test Management.

1 INTRODUCTION

This report discusses some of our experiences made in the area of software testing. These experiences cover mostly test automation and the architecture of testware, and, to a lesser degree, requirements engineering and design for testability. The terms test automation and automated testing in this context refer primarily to the automation of the test execution and support for test management or closely related tasks. Test automation in this paper does not cover the automated generation and validation of test cases and test results. The experiences have been made in a dozen projects during the past three years. For the five most important projects, we give a brief outline (Section 2), for subsequent illustration and as a basis for qualitative analysis.

The authors have been involved in these projects in various roles: software architect, software engineer, test consultant, test manager as well as tester. We have observed and analyzed our own mistakes and those of the other team members. What we found to be the six most interesting observations together with the potential rationale behind them, is discussed in Section 3. Based on this discussion, we present a table summarizing our findings, as well as four major lessons learned as a key to successful automated testing (Section 4).

We do not claim that our observations, experiences and consequences are the most important ones or even exhaustive. This paper is an experience report. The findings are based upon the consolidated experience of the authors, and are not the result of one ore more controlled experiments. Hence, they may or may not be applicable to other projects. However, in most of our reference projects they played a major role. Overall, this report is intended to validate, from a practical point of view, current day approaches in automated testing and what is anticipated to be good testing practice.

2 OVERVIEW OF PROJECTS

This section gives a brief overview of five of the projects on which the observations and experiences are based. They are representative in that they come from different application domains and test automation in its various facets plays an important role.

Project A: System to Manage Distribution of Assets

In this project, the client was in the process of the rollout of a large number of hardware assets (desktops, laptops, monitors, etc.). An operative asset management system was – among a

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.

* The third author is also an Adjunct Professor at the CS Department at Université de Montréal, Canada.

couple of other things – the prerequisite for this rollout. This system was intended to keep track of the delivered hardware and their configuration in order to feed billing systems. After failed attempts to get the system into production, test specialists should identify the problems and help to get the system productive. The engagement objective was to establish and develop an effective quality assurance and testing for a browser based asset management system. Main goals were to identify and isolate the issues, which directly hampered the productive operation. The testing work has been performed under specific circumstances with respect to a distributed development, an incomplete application, incomplete application documentation and very tight time lines – as said above the rollout was dependent on the system.

Project B: Java-Based Application Platform

The purpose of this project was the development of an application platform to standardize the development and operation of java- and J2EE-based applications, respectively. Part of the engagement in this project was an assessment of the quality of some framework components (e.g. for logging, auditing) of the platform and an assessment of the current status concerning development, build and test practices.

The analysis was followed by the design and realization of a family of reference applications to test the components of the platform. This included documentation in form of (so called) cookbooks, which showed how to build (and test) these applications, which are platform conform. The reference applications together with automated functional tests were used to analyze the quality of the platform components. In order to get a comprehensive notion about 'which parts of the platform where touched by the analysis', tools to measure test coverage [5] and profilers [7] were used heavily.

Project C: Point of Sale System for Life Insurances

The system to be developed was a distributed point of sale (POS) system, developed for the life insurance division of a large bank [1]. It was intended to assist the sales process of life insurances. This bank intended to enter the life insurance market with a complete portfolio of products. Instead of building a new sales organization, the existing infrastructure (branch offices) should be used. Therefore, the POS system should enable finance people and bank clerks to offer and sell life insurances to bank customers.

Requirements engineering was based on use cases and on explorative prototyping. The POS system was realized in a common multi-tier architecture with a relational database in the background, CORBA as middleware, and Java as the main language to realize both the business components on the middleware, and the client components in form of a thin client.

Project D: Sales Support for Tailored Industrial Facilities

Our assignment was to build a family of intranet based applications supporting the sales department of a large international company in the industrial sector. The main goal was to create offers for customized industrial facilities within a short time.

In this project, an XP-like development process was used with strong involvement of the customer. Flexible responses to changing requirements and very short release cycles were essential to the customer, as he had to cope with conflicting and changing requirements from different national sales departments.

The test strategy therefore emphasized on automated tests at different levels, employing techniques supporting effective test automation like mock objects, and on daily integration combined with the execution of all automated tests. Testware architecture was considered as a part of the system architecture by the development team. At the end, test automation code accounted for about 25% of all code in the system.

Project E: System Test Automation for Control System

In this project, our client was suffering from long release cycles of his safety critical control and information system. The application was designed as a distributed system that could be tailored to match customer needs by configuration. The distributed components communicated via a message bus. The system supported different hardware platforms, as well as a configurable look and feel of the user interface.

A complete regression test took almost three months, due to the fact that tests had to be repeated on different hardware configurations. Much emphasis was put on the verification of the failover mechanism, which was at the heart of the system.

Our assignment was to automate an existing smoke test suite, which had to be run on two different hardware configurations once a week to verify the stability of baseline builds. Another goal of the automation project was to design and implement a testware architecture that could be reused in the automation of a regression test suite later.

3 OBSERVATIONS AND EXPERIENCES

3.1 Test Automation Strategy Is Often Inappropriate

A sound testing process is helpful for successful test automation, but an appropriate test (automation) strategy is vital. The test strategy defines, which test types – e.g. functional tests, performance tests, reliability tests – are to be performed on which test level – e.g. unit, integration, etc. – and which tests are automated and/or supported by tools. Four common mistakes with regard to test automation strategy are listed below:

Misplaced or Forgotten Test Types

Tests that are hard to do manually very often are hard to automate as well. Tests situated in the wrong test level usually are hard to execute, regardless whether they are executed manually or automatically.

Tests on different test levels usually have different goals. Unit tests generally focus on the program logic within a software component and on correct implementation of the component interface. It would be very inefficient to test these issues with a GUI based system test approach. It may be difficult to force the system under test into system states needed. Or the resulting

system state cannot be verified accurately, because it is not visible on the user interface. Additionally, the debugging effort for program logic bugs detected during system tests will be considerably higher.

Many organizations, we have been working in, rely mainly on system tests, with only unsystematic unit testing. Integration tests usually are completely neglected. This leads to inefficient testing. Moreover, some aspects like robustness tests, that are notoriously hard to test, are usually omitted completely.

Wrong Expectations

Many organizations have unrealistic expectations about the benefits of test automation. Test automation is intended to save as much money as possible spent for 'unproductive' testing activities. They therefore expect a very short ROI on their test automation investment. If these expectations are not met, test automation is abandoned quickly.

There are a few points in test automation that are not that easily incorporated into ROI calculations, but are strong points for automating tests:

1. Automated testing does allow for shorter release cycles. Test phases can be shortened considerably. Tests may be executed more often, bugs are detected earlier and costs for bug fixing are reduced.
2. The quality and depth of test cases increase considerably, when testers are freed from boring and repetitive tasks. They have more time to design more or better test cases, focusing on areas that have been neglected so far.

Missing Diversification

Organizations new to test automation usually have a very clear goal: saving time, money and most important scarce testing resources. The most natural way to achieve this goal is to automate whatever the testers have been doing manually so far. Consequently, many organizations start by automating some subset of their existing GUI based system tests. Frequently, this is not an efficient strategy for test automation:

- Developing test cases interacting with a GUI is usually very time consuming. Graphical user interfaces tend to change frequently and test scripts have to be adapted to these changes.
- Verification of system response and system state may be hard too, as the system state may not always be visible, or because verification of the system state makes the test case even more prone to high maintenance cost.
- Automated tests on user interface level tend to find only the failures the test designer intended them to find. Fewster and Graham stated in [8], that a tradeoff between test sensitivity and robustness is necessary in most circumstances.

A good automation strategy therefore *combines different approaches* for test automation: system test automation, integration test automation, and unit test automation, which is usually most effective. In many cases, test code implemented for one approach can be reused for other test types or approaches, thus making a combined strategy even more effective.

Tool Usage is Restricted to Test Execution

Strategies for automated testing often consider only automation of test execution. Sometimes there is more potential in automating processes in the test lab like installation and configuration procedures. Additionally, tools may be used to design test cases or test reports more efficiently. The same is valid when it comes to analysis and reporting. The appropriate and consequent usage of a good test and change management tool often and easily saves more than the mere automation of the test execution. These areas are often overlooked, when a test strategy is defined.

Project References

In project C, the automated test (execution) nearly failed due to wrong expectations. This time from the developer side; it was expected to detect more errors through the mere execution of the test suite. However, the strategy was sufficiently diversified and the automated functional tests were partially reused to drive a performance test against the business logic of the system. Automated functional test were based on JUnit and for the performance test, a decorator based on JUnitPerf [6] had been used. This saved considerable effort compared with conventional, GUI-based performance testing.

In project E, complete automation and sufficient diversification was not possible due to limitations in the system architecture. Automated test where restricted to the GUI-level. Most of the test cases in the smoke test suite in project E were concerned with verifying that the failover mechanism worked correctly. Both simulating a failure and verifying the system response to this failure was not always possible. In many cases, it would have been easier if the automated tests had interacted directly with the system components instead of interacting with the user interface only. However, despite these conceptual problems scarce testing resources could be freed from running the smoke test suite manually once a week. And most important, the weakness in the automation strategy has been recognized and will be addressed in future development cycles.

A different strategy could be applied in project D. The tests were not restricted to script-based automation of GUI-based tests. Tests were executed on different levels: on module, integration and system test level. Much emphasis was on early unit tests, which accounted for more than 50% of the test automation code. Automated integration test techniques were applied to test the interaction between the business layer and the data access layer, *without* interacting directly with the user interface. System tests were restricted to the verification of the most important functions only, and the system test automation code only accounted for about 20% of all test code. Test cases were reused for load and volume tests. A suite of manual system tests, which lasted for about one day, complemented the automated system tests.

3.2 Tests Are Far More Often Repeated Than One Would Expect

For some test types – e.g. load and performance tests – testing is simply not effective without automated tests and proper tool support. For other test types – e.g. functional tests or tests of the error handling – the cost-effectiveness of automated tests depends highly on the number of times the test is executed. Each time an automated test case is executed without manual intervention its potential cost-effectiveness grows.

In general, test cases are executed (1) during the immediate lifetime of a project, and (2) if there is any next release – usually there is one – survive the project and are executed in following releases. Hence, test cases usually survive the first project. In our experience, it is usually underestimated for both cases and by a large number how often a test case will be executed.

If test automation is not cost effective, it is rarely because the number of (estimated) test executions is too small to justify the automation. When the number of repeated test executions is an argument for the cost effectiveness of automated tests, the threshold for the break even point is lower than commonly expected. If automated testing is not cost effective the cause are seldom missing repetitions, but an inappropriate test automation strategy (see Section 3.1) or an application architecture, which is not designed with testability in mind (see Section 3.5) or despicable testware architecture (see Section 3.6).

If you expect a test case to be executed more than ten times it is a potential candidate for an automated test case. In our experience, almost all test cases are executed at least five to ten times and at least 25% of the test cases are executed by far more than 20 times.

Given a sound test strategy, we observed and experienced the average overhead to automate a given manual test case a little below factor 2. However, the variance is high and an overhead up to factor 30 can be observed easily. So, it is wise to choose carefully. For successful test automation, it is key to start with the automation of the test cases that promise the highest return on investment. Candidates which usually are run quite often are

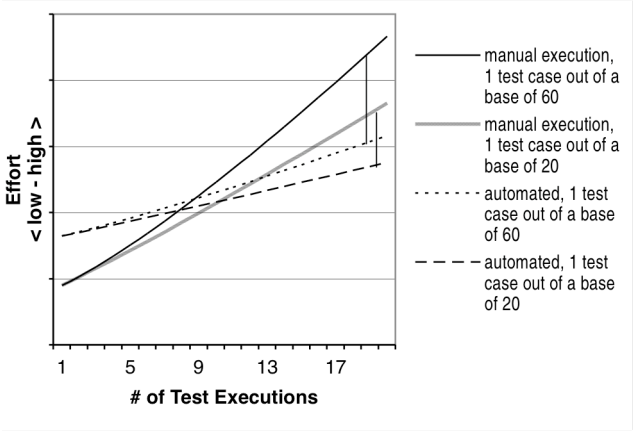

Figure 1. With an increasing number of test cases, the effort for manual execution becomes *disproportionally* higher in comparison to the automated variant.

smoke tests, component and integration tests. As indicated above (see Section 3.1), extreme caution is needed if script-based automation of GUI based tests is attempted. These test require more effort to create ([9] they start with factor 3), deliver a large number of false positives, are difficult to maintain, and are therefore executed less often than expected. Kaner states in [8] that GUI regression testing leads to weak design of the testware and we share this experience.

Another experience is to start small, execute automated tests very often, learn from the experience, and improve as you go on with test automation. A structured bottom-up approach is in this context more appropriate than a top-down one.

Project References

Testing in project A started with a very short timeline. Tests were a combination of user interface tests and tests of import and generation of data feeds for other systems. Due to the short timeline and the potential risks, it was decided not to attempt an automated approach. The overall approach could have been characterized as 'we do not have the time to be efficient'. It was implicitly assumed that the number of execution would be too small – three to five complete cycles.

In retrospection, the number of estimated number of executions was by factor 8 below the real numbers. A complete manual regression cycle with around 90 test cases took three persons (two testers, one developer) around one week. Three and a half days of this week were pure test execution, one and a half day were analysis and reporting. Later in this project, a moderate automation of both execution and reporting was attempted and reduced the complete cycle to less than 3 days. Another observation in the context of automation and comparison of automated and manual test cases was that with an increasing number of test cases the effort for manual execution of one test case is disproportionately higher in comparison to the automated variant (see Figure 1). We suspect a cause for this to be the increasing effort to keep the test cases consistent.

3.3 The Capability To Run Automated Tests Diminishes If Not Used

Test automation often fails because the automated tests are not run often and frequently enough. The test suite degrades into a state where the test cases are inconsistent and difficult to understand. It is not uncommon that this is the main cause for the failure of automated testing – even with an appropriate strategy and testware (see sections 3.1 and 3.5). This is not a strategic problem but 'only' an operational one. Automated test suites have a strong tendency to be quite unforgiving when suspended and not run for a short or even very short time. The effort to run and maintain an automated suite increases by a disproportional amount if not run frequently. We often observe a common failure pattern in four phases (see Figure 2).

Phase 1

Automated test cases are created in the detailed design or early implementation phases of a project. The tests are not only a welcome tool to find defects but also help to understand the

domain problems to a greater degree. Hence, their perceived necessity not only in terms of defect detection is high. As there are a relatively small number of automated test cases in this phase, the understandability and integrity is very high. This means the test can be run automatically and there are a small number of false positives, i.e., erroneous test cases. False positives are fixed immediately. Projects with an inappropriate test strategy often fail early – at the end of phase 1 or beginning of phase 2 because the development of the test cases does not improve domain understanding, the development of the test cases takes too long, and the quality of the test cases in terms of (initial) defect detection is not sufficient.

Phase 2

The development team is now fluent with the tools and as new functionality is added to the system, many new test cases are added rapidly. In our experience, most new defects are detected during the initial creation of automated test cases, and *not* during their repeated execution. Hence, it is observed by the team members that the existing test cases have a lower benefit in terms of error detection. However, they have a high benefit in terms of a prevention of error (re-)introduction, but this often is not or cannot be seen at this stage and it is also not that important at this stage.

The test suites are still small – yet rapidly growing – and because they are still understandable due to their small size, false positives often are not fixed immediately. For obvious reasons, it seems more fruitful to create new test cases rather than fix or maintain the existing ones. The team members assume that it is 'normal' that some test cases are always 'red' and usually they still know why. The perceived necessity for automated test cases drops further as the team members become more proficient with the domain. Less effort is spent for the maintenance of the existing test cases. With less maintenance, the knowledge about what these test cases exactly do and the trust slowly vanishes (cf. [12]). By the end of phase 2 the test suites are neither integer nor sufficiently understood. It is more convenient not to run the automated test, at least not too often and not completely.

Phase 3

As stability of the interfaces and reliable contracts becomes more important, the perceived necessity for automated test grows again.

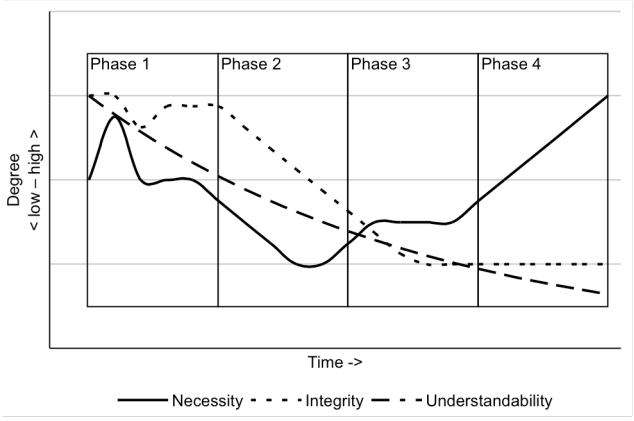

Figure 2. Necessity, integrity, and understandability of automated tests over time.

The benefits of daily execution of automated tests – faults are detected earlier, and shorter release cycles because of shorter test phases – would be welcome now. However, the understandability and integrity of the automated test suites are too low for an effective execution and it cannot be restored without considerable effort. At this stage of the project, it is often simply not possible to (re-) invest the necessary effort into (re-) establishing the operability of automated test suites. If it is not done, the integrity and understandability drops below a level where a restoration of the operability of automated test suites is no longer possible and economic, respectively. Mid of phase 3 is usually the last possibility to restore the operability of an automated test suite.

Phase 4

The capability to run automated tests has been lost in the project. Unfortunately, this is often a phase where the capability to run automated regression tests would be most welcome – often because there are a lot of integration, deployment or performance tuning activities. The perceived necessity for automated test grows again. If it reaches a sufficient level, yet another attempt for automated testing is started.

Project References

Project C was a good example for a project, which followed the first three phases. In phase 3, it was decided to reinvest the effort to make the automated test operational again. The background was here, that it was possible to reuse some of the tests for a performance test – a good argument to justify the effort.

To raise the understandability of the test suite to an acceptable level the test cases were reorganized along test scenarios, which were derived from the use cases. This simplified interpretation and analysis of a failed test case considerably.

3.4 Automated Testing Can Not Replace Manual Testing

It is the manual tasks that detect most new defects and not the automated ones. Kaner states in [9] that of the bugs found during an automated testing effort 60% - 80% are found during the development of the tests. We *strongly* support this observation.

Running an automated test is not really a destructive kind of testing in the sense of Myers 'falsify the software with respect to stated and unstated requirements' [10]. An automated test *re-validates* a unit under test. An automated test is constructive and not destructive by nature. It cannot find defects an experienced tester reveals. Good testers use their knowledge of weaknesses in the development team and in the technology to provoke failures and detect errors. It is not the repetition but the development of an automated test and its initial execution that reveals most defects. The replay of automated tests infrequently reveals a new defect; it detects the introduction of similar defects – defects that already occurred before.

In this sense, test automation often sets false expectation and a treacherous illusion of good software quality when it is expected that a simple and straight replay of test cases detect the defects.

With automated tests, the expert testers are freed from running the same boring regression test suite over and over again and more resources are available for difficult tasks. Automated tests facilitate the use of coverage tools and profilers, which significantly eases an iterative refinement of test cases. The feedback loop between developing a test case, executing the test case and assessing the quality of the test case (in terms of, which parts of the code are touched) is immediate. This is far more difficult with manual tests. It helps to create better test cases, with a defined quality with less effort.

Project References

In project B, automated tests were developed primarily to assess certain mostly non-functional qualities of selected framework components. The tests were JUnit tests based on functional specs. The purpose was not test automation with the objective to have a large functional regression suite. Instead, the objective was to have an in-depth validation of more non-functional qualities like error handling, testability or operability.

The tests were developed iteratively with the usage of tools to measure coverage to prove that the right code is touched. The development style was not a typical approach to develop automated tests, but like a specific approach to the iterative development of repeatable tests, which can also be executed automatically. In short, the steps described in the test development cycle were as follows: (1) Document the objective of the test case, (2) develop/refine automated test case, (3) execute test case, (4) measure coverage (statement and branch), and (5) validate the result against the objective. Refine test case if necessary (proceed with step 1 or 2).

In retrospection, this turned out to be an effective and convenient way, which supported a focused development and a concentration on the essence (in the sense of Brooks [2]), because it helped to construct test cases that reached the code that is difficult to address – especially the error handling parts. It was easy to realize, whether the intended parts of the code had been touched;

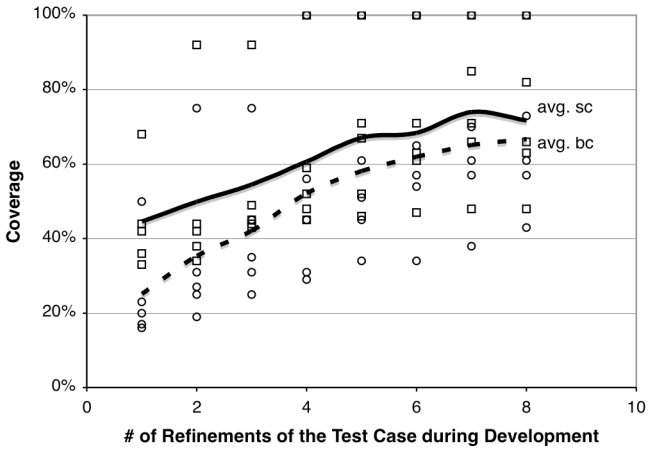

Figure 3. Increment of statement (sc) and branch coverage (bc) during the iterative development of a test case. Circles represent individual measurements of the sc and squares individual measurements of the bc.

the tool to measure the coverage [5] marked the reached statements in the editor.

We tracked the increase in statement and branch coverage for five test cases over eight refinements. No (non-trivial) test case initially reached more than 50% statement and 30% branch coverage (see Figure 3). This shows that it is incredibly difficult to deliver a good test case right from the start. The vast majority of the defects were detected in the cycle of test case development, especially in the refinement steps. Although the tests were run on three different releases of the platform components new defects were scarcely detected. The redetection of known defects was below the expectations too.

3.5 Testability Is A Usually Forgotten Non-Functional Requirement

We consider testability to have two distinctive aspects: (1) The *design of the software itself for testability* and (2) the *design of environment for the automated testing*. It is obvious that the latter plays an important role with automated testing. Nevertheless, the first is more or at least equally important for automated testing, because it can considerably increase or decrease the effort to create automated test cases. We consider it the primary cost driver and thus the main enabler or disabler for automated testing.

Design for Testability

Quite often systems are difficult to test not because it is difficult per se, but because the architecture of the system makes it cumbersome. More technically speaking this means, systems with:

- No visible or usable layering that allows independent testing of parts of the system separately.
- Not enough possibilities to mock parts of the system, in order to allow for an isolated test of layers and a systematic test of error conditions.
- Too little checked assertions to allow for self-diagnosis and proper state of error conditions.
- Incomprehensible or missing error messages.

One can explain this situation with the usual arguments 'no (time|money|people) to be efficient' or 'not invented here', but beyond this, the reason for this must be sought mainly in two areas: (1) *missing requirements* and (2) *inadequate design for testability*.

(1) Frequently, the corresponding requirements are not in the specification; either they have been simply forgotten or intentionally left out. Many specifications are mostly confined to functional (business) requirements, which are of course and by nature those the client is most interested in. Requirements concerning testability – like operability, deployment or the adequacy of notifications, error messages and/or log entries – are missing. The obvious consequences are, that systems are seldom designed to cover these missing requirements – why should they?

(2) Design for testability: it is not really explored what a testable architecture is. We think of testable architectures in terms of *layers*, *mocks*, *assertions*, and *adequacy of notifications*, but we

know of few directly related publications like [11] dealing directly with design for testability. There has been done a lot of work about maintainable (meaning change-friendly) architectures, which is somehow related. However, there are many additional aspects, which are not covered, and there are some aspects that are contradictory; in these cases, explicit decisions have to be taken, where maintainability or other qualities is related to testability.

Both factors have an impact on automated testing and an impact on how systems are tested in general. The impact on automated testing is that an inadequate design de facto enables or disables automated testing. In contrast to the automation strategy or the design of the testware, which can often be corrected, it is often impossible to correct the design of the software under test, if it inhibits cost effective automated testing. The impact on testing in general is, that the way a system handles errors is insufficiently tested. For most of the (commercial) software we have seen, we would claim that significantly less than 50% percent of the code that handles exceptions and errors has ever been executed before the system is integrated. The reason for this lies in the way tests cases need to be developed for error handling code; usually deeply nested structures have to be reached. Without tool support to measure coverage and the possibility to run and refine the corresponding test cases, it is difficult and time consuming to create the appropriate test cases (see Section 3.4).

Design of the Test Environment

Usually, fussiness in the design of the test automation environment has not the same big impact on automated testing as the design of the software under test or an inadequate test strategy, as it can be corrected more easily. The most common problems we have experienced are (1) manual installation and configuration procedures for the system under test, (2) lack of access to essential infrastructure like the configuration management system, and (3) an automation where the test execution is not observable (enough) to the tester.

Project References

Due to the absence of a usable layering in project B, the most effort in developing the test cases went into exploring the unit under test in order to find out if or how a test case can be automated. As mentioned before, the purpose was not test automation, but the assessment of certain qualities. In general, the initial test case was inappropriate and without measurements and further refinement, it would have been ineffective to reach the goal. The construction of test cases, which covered the error handling code sufficiently, turned out to be extremely demanding. For many error conditions, it was impossible to build test cases that reached the corresponding code. Responsible was an architecture not designed with testability in mind.

Test automation in project E was not as efficient as expected. However, it was (cost) effective in the end because it was a smoke test, which is predestined for automation, because it is executed often. The GUI-based scripting approach was hampered by several issues related to testability both in the test environment (manual deployment process, simulators for external systems designed for operation through the user interface only, lack of access to the configuration management system) and in the system under test (log files not suited for automated parsing, only rudimentary support for GUI library used in common GUI test automation tools), resulting in a very high effort for test automation: approximately 4 days for one test case with 40 test cases, which amounts to 160 days. A manual run took 2 days, the return on investment (maintenance not considered) was approximately 2 years.

3.6 Testware Maintenance Is Hard

Testware comprises everything needed for (automated) testing, including for example test scripts, test drivers, simulators, but also input data, expected output, utilities and data used to initialize the test system and utilities for creating test protocols and reports.

Testware has to be maintained with each new release of the system under test. Depending on the frequency of release cycles and the lifespan of the system under test, its maintenance tends to have a much bigger impact on the overall cost for testing than the initial implementation of automated tests. Testware architecture therefore has to be designed to minimize maintenance cost.

Weinberg's *Pattern Zero* [13] is frequently visible – a Pattern Zero organization is oblivious to the fact that it is actually developing software. Test automation projects often are done without proper design, planning and documentation of testware architecture. This has severe consequences on the effort needed to maintain test scripts and automation infrastructure.

Undocumented Architecture

Test software is usually not engineered with the same diligence as it would have been done in a 'real' software project. Corporate software development standards are often neglected, important architectural decisions are taken ad hoc during implementation. The initial architecture tends to degrade very fast.

Missed Opportunities for Reuse

Tests tend to be repetitive. The same interactions with the test object are repeated over and over again. With a naive test automation approach, those repetitive actions are reimplemented, resulting in high maintenance cost if the test object interface is changed. This causes serious problems in case of GUI based

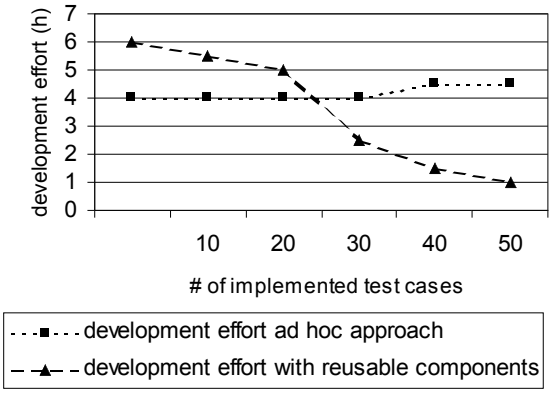

Figure 4. Development effort / test case (project D)

system test automation, because slight changes in the user interface, which are quite common, can lead to adjustments in many test scripts.

Another opportunity for testware reuse is utilities, drivers and simulators. Once available in one project, they may be reused in other projects. Other testware components can be obtained from the open source community or purchased from third party vendors.

Poorly Structured Testware

Reusable utilities and components will not be used if developers do not understand them. Most developers will spend at most a couple of minutes looking for something that is supposed to be there, before they reinvent it themselves. In order to avoid duplication of testware, it has to be structured and documented in a way that its components are easy to locate and use.

In a typical test project with several hundreds or thousands of test cases, it may become very hard to analyze a failed test and locate a script for a test case, if they are not organized in a clear and intuitive way. It is important to be able to identify test cases and scripts affected by changes in the system under test in order to adjust them in an efficient way.

Not only test scripts need to be maintained, but also test data. Typical symptoms of unstructured test data are (1) duplication of test data leading to increased maintenance effort or (2) unexpected side effects on other test cases, when maintaining test data for individual test cases or a group of test cases.

Untested Testware

Test code needs some amount of testing, too. It is very common for automated tests to detect an error when there is none or to miss the failures they were actually designed to detect, because of a bug in the testware. If reuse of test code is a goal, even well structured test scripts and utilities become quite complex. Any software utility or reusable component built for test automation should be tested at least superficially with an automated test suite. Test cases should be forced to fail at least once, in order to make sure it doesn't miss the point it was designed for. This is not always easy to do, because changes in the object under test may be necessary to trigger a test to fail.

Project References

In Project D, system tests initially were automated without giving much attention to testware architecture. After some 30 test cases have been implemented, the test maintenance used much of the resources allocated for system test automation, because the user interface was still frequently changing in response to changing customer requirements [3]. As a consequence, all existing test cases had to be redesigned. More emphasis was put on reusable test code modules, which encapsulated the business processes and their interaction with the user interface. These modules were reused in different test cases. With this approach, not only the effort for maintenance could be reduced considerably, but also the development effort for new test cases.

In the initial unstructured approach, the average implementation effort for one test case derived from use cases was about 4 hours. With the new approach focusing on reuse of test code, the effort increased to 6 hours at the beginning, dropping very quickly to not more than 1 hour per test case after 50 test cases had been implemented (see Figure 4). Higher development effort for reusable components at the beginning soon was outweighed by the benefits of reusing those components in other test cases.

4 SYNOPSIS AND CONSEQUENCES

In this section, we would like to do two things: First, give a kind of quantitative basis for the evidence of our observations and second, attempt to draw possible consequences in form of advice to circumvent the greatest cliffs in test automation.

Table 1 summarizes the observations and experiences discussed in Section 3. It gives an overview in which project we assume them to be valid, considering how often we observed them. However, we do not have data from controlled experiments to supply evidence for our observations and experiences. In order to gain more confidence we evaluated them with respect to the five given projects using an ordinal scale with four degrees: observed extensively, observed, not observed, opposite observed. *Observed extensively* means, that after revisiting the project history, the authors agreed that the corresponding observation was a typical and frequent pattern. *Observed* means that it occurred, but not as a typical pattern or only with minor consequences. *Not observed* means that the corresponding observation did not occur and not

Table 1. Support or contradiction of observations and experiences.

Observation / Project	3.1 Test Automation Strategy Is Often Inappropriate	3.2 Tests Are Far More Often Repeated Than	3.3 The Capability To Run Automated Tests Diminishes If Not Used	3.4 Automated Testing Can Not Replace Manual Testing	3.5 Testability is a Usually Forgotten … Requirement	3.6 Testware Maintenance Is Hard
Project A	Observed extensively	Observed	n/a	Observed extensively	Observed extensively	n/a
Project B	n/a	Observed	Observed extensively	Observed extensively	Observed extensively	Observed
Project C	Observed extensively	Observed extensively	Observed extensively	Observed extensively	Observed	Not observed
Project D	Opposite observed	Observed	n/a	Observed	Opposite observed	Observed extensively
Project E	Observed extensively	Observed	n/a	Observed	Observed extensively	Observed

caused any problems, although it could have by the nature of the project. In projects where the problem had been explicitly addressed and (successfully) managed, *opposite observed* is indicated. A *n/a* is indicated, where we had not enough insight in the project for a qualified answer or where the observation was impossible due to the nature of the project.

The strongest support has been found for the observations discussed in sections 3.2 and 3.4. Observation 3.3 could be observed in two projects (B, C) only. In the other projects, a conclusion was not possible; mostly, because we were not long enough in those projects, the automated tests were not established long enough, or because the test suite was run every day. Nevertheless, we found it important and observed it many times in projects not mentioned in this report. For the observations 3.1 and 3.5, the opposite was observed in project D. Despite this, we agreed this to be valid often enough to be mentioned anyway and concluded that projects where the strategy is right might have a higher chance to work toward testable designs. A (welcome) exception was project D, where much emphasis was put on a good test automation strategy from the beginning, and where testability requirements could straightforwardly be considered in the XP-like development process with developers also responsible for testing activities.

We have summarized our observations in four consequences, which, in our opinion, help to successfully realize sustainable test automation projects if considered carefully.

Adopt a Sound Test Strategy

A sound test automation strategy is crucial to test automation success and the only way to avoid problems, which arise from observation 3.1. An explicit test strategy gives the possibility to estimate cost effectiveness beforehand (see observation 3.2). We suggest organizations considering test automation to proceed in four steps:

1. Define a testing strategy, which takes into account your quality objectives and the specific characteristics of your test object. Integrate activities to constantly refine and maintain test cases.
2. Define goals for test automation. These goals may be shorter release cycles, saving money or better product quality, or any combination of these or other goals.
3. Choose a diversified test automation approach. Strategies combining several approaches are usually more effective than those only using one single approach.
4. Frequently evaluate your automation approach. Use appropriate measurements to evaluate whether your test automation goals have been achieved and constantly improve your automation.

Design for Testability

Whereas an inappropriate test automation strategy may be improved gradually, it may be very difficult or even impossible to change a system architecture not suited for automated testing. Consider testability in your system architecture right from the beginning. The most serious problems arise from observation 3.5 and 3.1. A good way to cope with these is (1) to feed your requirements engineering process with non-functional requirements derived from your test (automation) strategy and (2) in terms of architecture of a system to enforce and focus on exploitable layers, mock-ability, assertions and adequacy of machine-generated notifications.

Integrate your Software Daily

In order to prevent your test automation regime from degrading gradually over time, integrate your software at least once a day and try (hard) to run all automated tests during integration. This is a simple and effective technical measure which greatly helps to avoid the problems which arise form observation 3.3 and 3.4. Failures revealed by automated tests should be fixed immediately, regardless of whether they are related to faults in the testware or in the object under test.

Apply Good Engineering Practices

Test automation projects are software development projects like any other, and they are prone to the same problems, like any other. In order to be successful, apply the same diligence and good software engineering practices as in other development projects.

Maintenance of automated test suites became a major burden to most organizations we have been working with (see 3.6). Where this was not an indirect consequence of the application architecture, it was caused by inappropriate testware architecture, poorly designed without consideration of maintainability issues.

5 REFERENCES

[1] Berner, S. About the Development of a Point of Sale System: an Experience Report. Proc. of ICSE 2003, Portland, OR, May 2003.

[2] Brooks, F. P. No silver bullet – essence and accidents of software engineering. Computer, 20(4), Apr 1987.

[3] Elkoutbi, M., I. Khriss, and R. K. Keller. Automated Prototyping of User Interfaces based on UML Scenarios. J. of Automated Software Engineering. To appear.

[4] Fewster, M., D. Graham. Software Test Automation: Effective use of test execution tools. ACM Press, 1999.

[5] http://www.cenqua.com/clover/

[6] http://www.clarkware.com/software/JUnitPerf.html

[7] http://www.quest.com/jprobe/

[8] Kaner, C. Architectures of Test Automation. Software Testing, Analysis & Review Conference (Star) West, San Jose, CA, Oct 2000.

[9] Kaner, C. Improving the Maintainability of Automated Test Suites. Software QA, 4(4), 1997.

[10] Myer, G. J. The Art of Software Testing. New York: Wiley & Sons, 1979.

[11] Pettichord, B. Design for Testability. Proc. of Pacific Northwest Software Quality Conference, Oct 2002.

[12] Pettichord, B. Seven Steps to Test Automation Success. http://www.pettichord.com (Revised version of a paper presented at STAR West, San Jose, Nov 1999).

[13] Weinberg, G. M. Quality Software Management: Systems Thinking, 1. Dorset House, 1999.