HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Software Business and Engineering Institute

Pekka Laukkanen

Data-Driven and Keyword-Driven Test Automation Frameworks

Master's thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.

Espoo, February 24, 2006

Supervisor:	Professor Reijo Sulonen
Instructor:	Harri Töhönen, M.Sc.

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

Author:	Pekka Laukkanen		
Name of the thesis:	Data-Driven and Keyword-Driven Test Automation Frameworks		
Date:	February 24, 2006	**Number of pages:**	98 + 0
Department:	Department of Computer Science and Engineering	**Professorship:**	T-76
Supervisor:	Prof. Reijo Sulonen		
Instructor:	Harri Töhönen, M.Sc.		

The growing importance and stringent quality requirements of software systems are increasing demand for efficient software testing. Hiring more test engineers or lengthening the testing time are not viable long-term solutions, rather there is a need to decrease the amount of resources needed. One attractive solution to this problem is test automation, i.e. allocating certain testing tasks to computers. There are countless approaches to test automation, and they work differently in different contexts. This master's thesis focuses on only one of them, large-scale frameworks for automated test execution and reporting, but other key approaches are also briefly introduced.

The thesis opens its discussion of test automation frameworks by defining their high-level requirements. The most important requirements are identified as ease-of-use, maintainability and, of course, the ability to automatically execute tests and report results. More detailed requirements are derived from these high-level requirements: data-driven and keyword-driven testing techniques, for example, are essential prerequisites for both ease-of-use and maintainability.

The next step in the thesis is constructing and presenting a framework concept fulfilling the defined requirements. The concept and its underlying requirements were tested in a pilot where a prototype of the framework and some automated tests for different systems were implemented. Based on the pilot results, the overall framework concept was found to be feasible. Certain changes to the framework and original requirements are presented, however. The most interesting finding is that it is possible to cover all the data-driven testing needs with the keyword-driven approach alone.

Keywords: test automation, test automation framework, data-driven testing, keyword-driven testing

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Pekka Laukkanen		
Työn nimi:	Aineisto- ja avainsanaohjatut testiautomaatiojärjestelmät		
Päivämäärä:	24.2.2006	**Sivuja:**	98 + 0
Osasto:	Tietotekniikan osasto	**Professuuri:**	T-76
Työn valvoja:	Prof. Reijo Sulonen		
Työn ohjaaja:	DI Harri Töhönen		

Ohjelmistojärjestelmien merkityksen sekä laatuvaatimusten kasvaminen aiheuttaa paineita ohjelmistojen testaukselle. Testaajien määrän lisääminen tai testausajan pidentäminen ei ole loputtomasti mahdollista, pikemminkin resursseja halutaan vähentää. Yksi houkutteleva ratkaisu on testauksen automatisointi eli osan testaustyön antaminen tietokoneiden hoidettavaksi. Erilaisia tapoja testauksen automatisointiin on lukuisia ja ne toimivat eri tavoin erilaisissa tilanteissa ja ympäristöissä. Tämä diplomityö käsittelee tarkemmin vain yhtä lähestymistapaa, laajoja automaatiojärjestelmiä testien automaattiseen suorittamiseen ja raportoimiseen, mutta myös muut tavat ovat tärkeitä.

Testiautomaatiojärjestelmien käsittely aloitetaan määrittelemällä niille korkean tason vaatimukset. Tärkeimmiksi vaatimuksiksi todetaan helppokäyttöisyys, ylläpidettävyys sekä tietenkin kyky automaattisesti suorittaa testejä ja raportoida niiden tulokset. Näiden vaatimusten kautta päästään tarkempiin vaatimuksiin ja todetaan mm. että aineisto-ohjattu (data-driven) ja avainsanaohjattu (keyword-driven) testaustekniikka ovat edellytyksiä sekä helppokäyttöisyydelle että ylläpidettävyydelle.

Seuraavaksi työssä suunnitellaan määritellyt vaatimukset toteuttava testiautomaatiojärjestelmä. Järjestelmän toimintaa sekä sen pohjana olleita vaatimuksia testataan pilotissa, jossa toteutetaan sekä prototyyppi itse järjestelmästä että automatisoituja testejä erilaisille ohjelmistoille. Pilotin tuloksien perusteella suunnitellun automaatiojärjestelmän voidaan todeta olevan pääperiaatteiltaan toimiva. Lisäksi kokemusten perusteella järjestelmään sekä alkuperäisiin vaatimuksiin esitetään joitain muutoksia. Mielenkiintoisin löydös on se että kaikki aineisto-ohjatut testit voidaan toteuttaa käyttäen ainoastaan avainsanaohjattua lähestymistapaa.

Avainsanat: testiautomaatio, testiautomaatiojärjestelmä, aineisto-ohjattu testaus, avainsanaohjattu testaus

Acknowledgements

This master's thesis has been done for a Finnish software testing consultancy company Qentinel mainly during the year 2005. I wish to thank my instructor Harri Töhönen, M.Sc. and all other Qentinelians for comments, feedback and patience.

From the Department of Computer Science and Engineering I first of all want to thank my instructor Professor Reijo Sulonen. Additionally I am grateful for Juha Itkonen and Casper Lassenius for their support and valuable comments.

I also want to express my gratitude to Mark Fewster who was kind enough to review the thesis in its early form. Mark's comments and positive feedback made me believe that the ideas I present are valid and the remaining hard work is worth the effort. Also I want to thank Petri Haapio who has been in charge of both the automation project where I got the original idea for this thesis and a new one where the automation framework presented in this thesis has been successfully taken into real use.

Finally, I would like to thank my family and my wonderful girlfriend Sonja for everything. Kiitos.

Espoo, February 24, 2006

Pekka Laukkanen

Contents

Terms

Acceptance Testing	A level of testing conducted from the viewpoint of the customer, used to establish the criteria for acceptance of a system. Typically based upon the requirements of the system. (Craig and Jaskiel, 2002)
Action Word	See *keyword.*
Actual Outcome	Outputs and data states that *the system under test* produces from test inputs. See also *expected outcome.* (Fewster and Graham, 1999)
Automation	See *test automation.*
Automation Framework	See *test automation framework.*
Base Keyword	A term defined in this thesis for *keywords* implemented in *a test library* of *a keyword-driven test automation framework.* See also *user keyword.*
Black-Box Testing	A type of testing where the internal workings of the system are unknown or ignored. Testing to see if the system does what it is supposed to do. (Craig and Jaskiel, 2002)
Bug	See *defect.*
Capture and Replay	A scripting approach where a test tool records test input as it is sent to the software under test. The input cases stored can then be used to reproduce the test at a later time. Often also called *record and playback.* (BS 7925-1)
Component	One of the parts that make up a system. A collection of *units* with a defined interface towards other components. (IEEE Std 610.12-1990)
Component Testing	Testing of individual components or groups of related components. (IEEE Std 610.12-1990)

Context-Driven Testing	A testing methodology that underlines the importance of the context where different testing practices are used over the practices themselves. The main message is that there are good practices in a context but there are no general best practices. (Kaner et al., 2001)
Control Script	See *driver script.*
Data-Driven Testing	A scripting technique that stores test inputs and expected outcomes as data, normally in a tabular format, so that a single *driver script* can execute all of the designed test cases. (Fewster and Graham, 1999)
Defect	Introduced into software as the result of *an error.* A flaw in the software with potential to cause *a failure.* Also called *fault* or, informally, *bug.* (Craig and Jaskiel, 2002; Burnstein, 2003)
Domain code	Part of the application code which contains system functionality. See also *presentation code.* (Fowler, 2001)
Dynamic Testing	The process of evaluating a system or component based on its behavior during execution. See also *static testing.* (IEEE Std 610.12-1990)
Driver	A software module that invokes and, perhaps, controls and monitors the execution of one or more other software modules. (IEEE Std 610.12-1990)
Driver Script	A test script that drives the test execution process using testing functionality provided by *test libraries* and may also read test data from external sources. Called *a control script* by Fewster and Graham (1999).
Error	A mistake, misconception, or misunderstanding on the part of a software developer. (Burnstein, 2003)
Expected Failure	Occurs when *a test case* which has failed previously fails again similarly. Derived from Fewster and Graham (1999).
Expected Outcome	Outputs and data states that should result from executing a test. See also *actual outcome.* (Fewster and Graham, 1999)
Failure	Inability of a software system or component to perform its required function within specified performance criteria. The manifestation of a defect. (IEEE Std 610.12-1990; Craig and Jaskiel, 2002)

Fault	See *defect*.
Functional Testing	Testing conducted to evaluate the compliance of a system or component with specified functional *requirements*. (IEEE Std 610.12-1990)
Feature	A software characteristic specified or implied by requirements documentation. (IEEE Std 610.12-1990)
Framework	An abstract design which can be extended by adding more or better components to it. An important characteristic of a framework that differentiates it from *libraries* is that the methods defined by the user to tailor the framework are called from within the framework itself. The framework often plays the role of the main program in coordinating and sequencing application activity. (Johnson and Foote, 1988)
Integration Testing	A level of test undertaken to validate the interface between internal components of a system. Typically based upon the system architecture. (Craig and Jaskiel, 2002)
Keyword	A directive that represents a single action in *keyword-driven testing*. Called *actions words* by Buwalda et al. (2002).
Keyword-Driven Testing	A *test automation* approach where test data and also *keywords* instructing how to use the data are read from an external data source. When *test cases* are executed keywords are interpreted by *a test library* which is called by *a test automation framework*. See also *data-driven testing*. (Fewster and Graham, 1999; Kaner et al., 2001; Buwalda et al., 2002; Mosley and Posey, 2002)
Library	A controlled collection of software and related documentation designed to aid in software development, use, or maintenance. See also *framework*. (IEEE Std 610.12-1990)
Non-Functional Testing	Testing of those requirements that do not relate to functionality. For example performance and usability. (BS 7925-1)
Manual Testing	Manually conducted *software testing*. See also *test automation*.
Oracle	A document or piece of software that allows test engineers or automated tests to determine whether a test has been passed or not. (Burnstein, 2003)
Precondition	Environmental and state conditions which must be fulfilled before *a test case* can be executed. (BS 7925-1)

Predicted Outcome	See *expected outcome.*
Presentation Code	Part of the application code which makes up the user interface of the system. See also *domain code.* (Fowler, 2001)
Quality	(1) The degree to which a system, component, or process meets specified requirements. (2) The degree to which a system, component, or process meets customer or user needs or expectations. (IEEE Std 610.12-1990)
Record and Playback	See *capture and replay.*
Regression Testing	Retesting previously tested features to ensure that a change or a defect fix has not affected them. (Craig and Jaskiel, 2002)
Requirement	A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents. Can be either functional or non-functional. (IEEE Std 610.12-1990)
Set Up	Code that is executed before each automated *test case* in one particular *test suite.* A related term used in *manual testing* is *precondition.* See also *tear down.*
Smoke Testing	A test run to demonstrate that the basic functionality of a system exists and that a certain level of stability has been achieved. (Craig and Jaskiel, 2002)
Software Test Automation	See *test automation.*
Software Testing	The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component. (IEEE Std 610.12-1990)
Static Testing	The process of evaluating a system or component based on its form, structure, content, or documentation. See also *dynamic testing.* (IEEE Std 610.12-1990)
System Testing	A comprehensive test undertaken to validate an entire system and its characteristics. Typically based upon the requirements and design of the system. (Craig and Jaskiel, 2002)
System Under Test (SUT)	The entire system or product to be tested. (Craig and Jaskiel, 2002)

Tear Down	Code that is executed after each automated *test case* in one particular *test suite*. *Test automation frameworks* run them regardless the test status so actions that must always be done (e.g. releasing resources) should be done there. See also *set up*.
Test Automation	The use of software to control the execution of tests, the comparison of *actual outcomes* to *predicted outcomes*, the setting up of test *preconditions*, and other test control and test reporting functions. (BS 7925-1)
Test Automation Framework	*A framework* used for *test automation*. Provides some core functionality (e.g. logging and reporting) and allows its testing capabilities to be extended by adding new *test libraries*.
Test Case	A set of inputs, execution preconditions and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. (BS 7925-1)
Test Oracle	See *oracle*.
Test Outcome	See *actual outcome*.
Test Runner	A generic *driver script* capable to execute different kinds of *test cases* and not only variations with slightly different test data.
Test Suite	A collection of one or more *test cases* for *the software under test*. (BS 7925-1)
Test-Driven Development (TDD)	Development technique where automated *unit tests* are written before the system code. Tests drive the design and development of the system and a comprehensive *regression test* suite is got as a by-product. (Beck, 2003)
Testability	A characteristic of system under test that defines how easily it can be tested. Consists of visibility and control. (Pettichord, 2002)
Testing	See *software testing*.
Testware	The artifacts required to plan, design and execute *test cases*, such as documentation, scripts, inputs, *expected outcomes*, *set up* and *tear down* procedures, files, databases, environments and any additional software or utilities used in testing. (Fewster and Graham, 1999)

Unit	A piece of code that performs a function, typically written by a single programmer. (Craig and Jaskiel, 2002)
Unit Testing	A level of test undertaken to validate a single *unit* of code. Unit tests are typically automated and written by the programmer who has written the code under test. (Craig and Jaskiel, 2002)
User Keyword	A term defined in this thesis for *keywords* constructed from *base keywords* and other user keywords in a test design system. User keywords can be created easily even without programming skills.
White Box Testing	Testing based upon knowledge of the internal structure of the system. Testing not only what the system does, but also how it does it. (Craig and Jaskiel, 2002)
xUnit Frameworks	Frameworks that ease writing and executing automated *unit tests*, provide *set up* and *tear down* functionalities for them and allow constructing *test suites*. The most famous xUnit framework is JUnit for Java but implementations exist for most programming languages. (Hamill, 2004)

Chapter 1

Introduction

Software systems are getting more and more important for organizations and individuals alike and at the same time they are growing bigger and more complex. It is thus only logical that importance of *software quality*[1] is also rising. Software faults have caused loss of huge sums of money and even human lives. If quality does not get better as systems grow in size, complexity and importance, these losses are only getting bigger. (Burnstein, 2003)

The need for better quality means more pressure for *software testing* and for test engineers taking care of it. *Test automation*, i.e. giving some testing tasks to computers, is an obvious way to ease their workload. Computers are relatively cheap, they are faster than humans, they do not get tired or bored, and they work over weekends without extra pay. They are not ideal workhorses, however, as they only find *defects* from places where they are explicitly told to search them and they easily get lost if something in *the system under test (SUT)* changes. Giving computers all the needed details is not easy and takes time. (Fewster and Graham, 1999)

Test automation can be used in multiple ways. It can and should be used differently in different contexts and no single automation approach works everywhere. Test automation is no silver bullet either but it has a lot of potential and when done well it can significantly help test engineers to get their work done. (Fewster and Graham, 1999)

This thesis concentrates on larger *test automation frameworks* designed for test execution and reporting. Before the scope can be defined in more detailed manner some

[1]New terms are *emphasized* when used for the first time and their explanations can be found from the list of terms on pages ix–xiv.

background information about different automation approaches is needed, however, and that is presented in Section 1.2. Even before that it is time to investigate a bit more thoroughly why test automation is needed and what are the main challenges in taking it into use.

1.1 Promises and Problems of Test Automation

A comprehensive list of test automation promises, as presented by Fewster and Graham (1999), is shown in Table 1.1. Similar promises have been reported also by other authors like Pettichord (1999), Nagle (2000) and Kaner et al. (2001).

Most of the benefits in Table 1.1 can be summarized with words efficiency and reuse. Test automation is expected to help run lots of *test cases* consistently again and again on different versions of the system under test. Automation can also ease test engineers' workload and release them from repeating tasks. All this has the potential to increase software quality and shorten testing time.

All these promises make test automation look really attractive but achieving them in real life requires plenty of hard work. If automation is not done well it will be abandoned and promises will never be realized. A list of common test automation problems, again by Fewster and Graham (1999), can be found from Table 1.2.

The general problem with test automation seems to be forgetting that any larger test automation project is a software project on its own right. Software projects fail if they do not follow processes and are not managed adequately, and test automation projects are not different. Of all people, test engineers ought to realize how important it is to have a disciplined approach to software development. (Kaner, 1997; Zambelich, 1998; Fewster and Graham, 1999; Pettichord, 1999; Kaner et al., 2001; Zallar, 2001; Rice, 2003)

1.2 Different Test Automation Approaches

This section briefly introduces main test automation categories as an introduction and background for the rest of this thesis. The focused scope and target for this thesis are defined in next section.

Run existing *regression tests* on a new version of a program	Being able to run previously created tests without extra effort clearly makes testing more efficient.
Run more tests more often	Automation means faster test execution which means more test rounds. Automation should also make creating new test cases easy and fast.
Perform tests which would be difficult or impossible to do manually	For example performance and stress tests are nearly impossible to conduct without automation.
Better use of resources	Automating repeating and boring tasks releases test engineers for more demanding and rewarding work.
Consistency and repeatability of tests	Tests are always run the same way so test results can be consistently compared to previous results from previous testing rounds. Tests can also be easily repeated on different environments.
Reuse of tests	Reusing tests from earlier projects gives a kick start to a new project.
Earlier time to market	Reusing tests and shortening test execution time fastens feedback cycle to developers. In the end that shortens the time to market.
Increased confidence	Running an extensive set of tests often, consistently and on different environments successfully increases the confidence that the product really is ready to be released.

Table 1.1: Common test automation promises (Fewster and Graham, 1999)

Unrealistic expectations	Managers may believe test automation will solve all their testing problems and magically make the software quality better. Automation experts should help managers setting their expectations right.
Poor testing practice	If testing practices and processes are inadequate it is better to start improving them than bringing in test automation. Automating chaos just gives faster chaos.
Expectation that automated tests will find a lot of new defects	After automated test has been run successfully once it is not very likely to find new bugs unless the tested functionality changes. Automators normally find more defects while they are developing tests than when tests are re-executed.
False sense of security	Just seeing a test report with no failures does not mean that the SUT did not have any. Tests may be incomplete, either not testing all features or not able to see failures when they occur. Tests may also have defects and show wrong results.
Maintenance	When the SUT changes also its tests change. Human test engineers are able to handle even major changes without problems but automated tests can fail after a slightest change. If maintaining test automation system takes more time than testing manually it will surely be abandoned. The same will happen also if adding new features to the automation system is too cumbersome.
Technical problems	Building and taking test automation system to use is a technical challenge which is unlikely to proceed without problems. Tools may be incompatible with the tested system and they also often have defects themselves.
Organizational issues	Successful test automation project requires both high technical skills and support from management. Test automation has also big impact on the organization and requires changes in many processes.

Table 1.2: Common test automation problems (Fewster and Graham, 1999)

1.2.1 Dynamic vs. Static Testing

In very high level testing and test automation can be divided into *dynamic* and *static testing*. In the former something is done to the tested system and test status is checked afterwards but in the latter the developed code is not executed at all. Examples of static testing are document and code reviews and static code analysis (e.g. syntax correctness, code complexity). Reviews are by their nature done mainly by humans while static analysis is mostly left for computers and there are plenty of good tools available. (IEEE Std 610.12-1990; Burnstein, 2003)

I believe static testing is important and can greatly help in finding defects even in early stages of projects. In this thesis, however, interest is only on dynamic testing.

1.2.2 Functional vs. Non-Functional Testing

Dynamic testing can be further divided into *functional* and *non-functional testing*. Aim of the former, as the name implies, is ensuring that the functionality of the system adheres to *requirements*. The latter is conducted to verify that other, non-functional, aspects of the tested system work as well. (IEEE Std 610.12-1990; Burnstein, 2003)

Generally all functional testing is done in two steps as illustrated Figure 1.1. First something is done to the tested system. Next *the test outcome*, outputs produced by the system and changes in system's state, is verified against predefined *expected outcome*. In most cases both of theses steps can be automated. (Fewster and Graham, 1999)

There is plenty to test other than the functionality. Burnstein (2003) mentions that other areas to cover include performance, security, usability, portability, re-

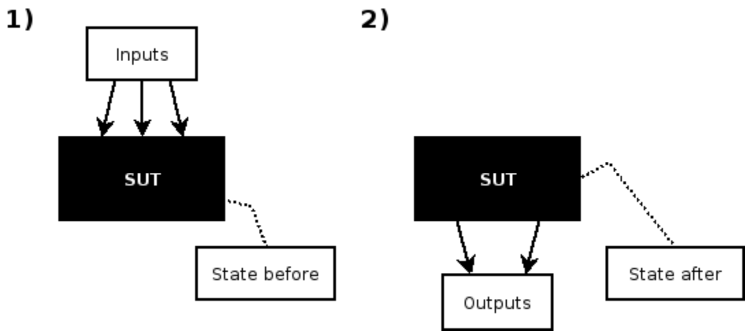


Figure 1.1: High level view to functional testing

liability and memory management. These testing areas are really important and failures in any of them may ruin otherwise good and functionally correct product. Non-functional testing differs quite a lot from functional testing and it is also rather heterogeneous itself. Some areas (e.g. usability) are mainly tested manually by experts while others (e.g. performance) are nearly impossible without adequate automation tools.

In this thesis the scope is on functional testing but many ideas presented are relevant also when automating non-functional testing.

1.2.3 Granularity of the Tested System

Traditionally testing is divided into different levels such as *unit*, *integration*, *system* and *acceptance testing* (Dustin et al., 1999; Craig and Jaskiel, 2002; Burnstein, 2003). In my opinion this division is not very practical in test automation context, however, because same automation approach can be used in many levels. For example system and acceptance testing do not have much difference from this perspective. That is why I prefer classification based on the granularity of the tested system into unit, *component* and system testing, as suggested by Meszaros (2003).

Unit Testing

The smallest building block of any system is a unit. Units have an application programming interface (API) which is used when interacting with other units and can also be used to test them. Unit testing is in most cases best handled by developers who know the code under test and techniques needed (Dustin et al., 1999; Craig and Jaskiel, 2002; Mosley and Posey, 2002). Unit testing is by its nature mostly automated. In some situations manual tests may be ok but, as noticed by Maximilien and Williams (2003), they are executed less often than automated ones.

Testing in unit level has multiple good characteristics. First of all unit interfaces can be driven without any special tools. In this level interfaces also tend to be more stable than in higher levels, especially in user interface level, and changes in other units have little or no effect unless units are closely connected. Finding and fixing defects is also cheapest in unit level.

De facto test tools in unit level are multiple *xUnit frameworks*. The first of the many was SmalltalkUnit (Beck, 1994) which was soon ported to Java as JUnit by Erich Gamma (Beck and Gamma, 1998). After JUnit's big success implementations for other languages—such as PyUnit for Python and NUnit for Microsoft's .NET—followed shortly. (Hamill, 2004)

Unit testing can be taken to new a level with *test-driven development (TDD)*. In TDD tests are, as the name implies, created before actual production code which forces developers to think about the design and *testability* of the system. As a result the design is clear and number of defects low (Beck, 2003). From testing point of view the nice thing about TDD is that a comprehensive regression test set is got as a bonus. Very promising results from using test-driven development are reported for example by Geras et al. (2004) and Maximilien and Williams (2003), and my personal experiences are also extremely positive.

The scope of this thesis is on higher test automation levels than unit testing. That does not mean that I would consider unit testing and unit test automation less important. On the contrary, I strongly believe that in most cases automation efforts should be first directed into unit testing level where needed investments are small and likely to pay back fast.

Component Testing

Components vary a lot from one situation to another but roughly speaking a component is a collection of related units which have a common interface towards other components. These interfaces may be similar programming interfaces as in unit level but they can also be implemented with many higher level techniques like COM, XML and HTTP.

Automation in component level is normally not too difficult. Test tools can be hooked into the same interface which is used by other components. In many cases there also exists good and free tools for driving the interface (e.g. HTTPUnit) and even if no ready-made tool is available developers can easily write one. (Pettichord, 2002)

System Testing

The difference between a component and a system is that systems function standalone while components are only used as part of a system. In real life this difference can be rather vague as systems are often integrated together to form even bigger systems.

Systems have different interfaces towards the rest of the world depending on whether they are going to be used by humans or other systems. Systems used by humans have some kind of user interface, either graphical or non-graphical, and systems used by other systems have similar interfaces as components. Graphical user interfaces are notoriously hard to automate but other interfaces are generally rather easy. (Pettichord, 2002)

1.2.4 Testing Activities

Software testing is much more that just executing test cases and similarly test automation is not limited to automating only test execution.

Designing Test Cases

Functional system and acceptance tests are designed by test engineers using system requirements and formal test design techniques like equivalence partitioning and boundary value analysis. Designing good test cases is not easy and it is one of the main skills a professional test engineer must possess. (Fewster and Graham, 1999; Craig and Jaskiel, 2002; Burnstein, 2003)

There are also ways to automate the test design process. Expected results for tests can sometimes be generated automatically using so called *oracles*, external trusted entities which can be queried for expected results. Often an oracle is some existing system, but they can also be created just for testing purposes (Richardson et al., 1992; Memon et al., 2000). Tools can also generate test cases based on software code or interfaces (Fewster and Graham, 1999). In model based testing the system is modeled in such a detail that test cases can be derived automatically from the model (Dalal et al., 1999; Fewster and Graham, 1999; Robinson, 2000). Plenty of other methods like a data mining approach (Last et al., 2003), a goal-driven approach (Memon et al., 1999) and a requirement-based test generation (Tahat et al., 2001) are also possible.

These kind of methods are very interesting but scope of this thesis is solely on test execution and task related to it. It should, however, be possible to integrate a test design automation tool to a test execution framework presented in later chapters.

Executing Test Cases and Analyzing Results

After test cases have been designed and created they can be executed and results verified as illustrated in Figure 1.1. Tests are often re-executed when a new version of the tested system is available. Automating these regression tests or at least an exemplary subset of them, often called *smoke tests*, can make test execution and result analysis considerably faster. (Zambelich, 1998; Pettichord, 1999)

Reporting Test Results

After test engineers have run their tests they report findings to the project team and management. If test execution is fully automated it makes sense to automate also reporting. It probably is not a good idea to send all test reports automatically to managers' mail boxes but if a test report is created automatically test engineers do not need to spend time gathering information from test logs and elsewhere.

Test Management

Many test management tasks like planning, monitoring, scheduling and defect tracking can be supported be tools. These tools and automation possibilities they provide are not in the scope of this thesis.

1.2.5 Small Scale vs. Large Scale Test Automation

The scale of test automation can vary from using it only in small parts of testing, like checking two files for equivalence, to large scale test automation systems doing everything from setting up the environment and running tests to reporting results. (Zallar, 2001)

Tool Aided Testing

Small scale test automation just helps manual testing. Test automation tools are used in areas where computers are better than humans and where tools are available or easily implemented. For example tasks like checking that an installation copied all files to right places and verifying that two files have same data take time and are error prone to humans but easily automated. This automation approach is strongly advocated by Bach (2003). He notes that in small scale automation the test code is generally so simple and inexpensive that it can either be easily fixed or trashed if it is broken by changes in the tested system.

Bach (2003) proposes an idea that so called toolsmiths work with test engineers and provide them with tools and utilities based on their needs. Toolsmiths should have good programming skills, at least adequate testing skills and deep knowledge of different automation tools and methods. There are plenty of good tools available for this and many of them even are free. Tools that I have used successfully myself include Unix originating utilities like grep and diff, shell scripts and higher level scripting languages like Perl and Python, and various tools built by developers for their development time testing. Bach (2003) and Kaner et al. (2001) list more tools and sources where to find them. As Kaner et al. (2001) puts it, you may already have more test tools than you realize.

Even though the scope of this thesis is in large scale test automation I believe small scale test automation can often be a better strategy because it does not require big investments and provides help fast. Especially if risk with larger frameworks feel too big it is better to start small, gain more experience and then later invest to larger frameworks (Zallar, 2001; Bach, 2003).

Test Automation Frameworks

When test automation is taken to highest level tests can be started with a push of a button, they can be left running over-night unattended and next morning test results are published. This kind of automation clearly requires some kind of a system which makes creating, executing and maintaining test cases easy. The system should provide some core functionality (e.g. monitoring and reporting) and allow extending itself to make it possible to create new kinds of tests. This kind of systems match the definition of *framework* by (Johnson and Foote, 1988) so it is appropriate to call them *test automation frameworks*.

Test automation frameworks have evolved over the time. Kit (1999) summarizes three generations as follows.

1. First generation frameworks are unstructured, have test data embedded into the scripts and there is normally one script per one test case. Scripts are mainly generated using *capture and replay* tools but may also be manually coded. This kind of script is virtually non-maintainable and when the tested system changes they need to be captured or written again.

2. In second generation scripts are well-designed, modular, robust, documented and thus maintainable. Scripts do not only handle test execution but also for example setup and cleanup and error detection and recovery. Test data is still embedded into the scripts, though, and there is one *driver scripts* per one test case. Code is mostly written manually and both implementation and maintenance require programming skills which test engineers may not have.

3. Third generation frameworks have all the same good characteristics already found from second generation. They go forward by taking test data out of the scripts which has two significant benefits. First benefit is that one driver script may execute multiple similar test cases by just altering the data and adding new tests is trivial. Second benefit is that test design and framework implementation are separate tasks—former can be given someone with the domain knowledge and latter to someone with programming skills. This concept is called *data-driven testing*. *Keyword-driven testing* takes the concept even further by adding keywords driving the test executing into the test data.

Third generation test automation frameworks are the actual interest of this thesis.

1.3 Scope

Previous section should have made it clear that test automation is a very broad subject and covering it all in a single master's thesis is impossible. Some large areas were already excluded from the scope in Section 1.2 and what was left is following.

> *Designing and implementing a large scale test automation framework for functional testing in component and system level. The framework must be able to execute tests, verify results, recover from expected errors and report results. It must also be both easy to use and maintain. It does not need to provide support for automation of test planning and design.*

Knowing how to automate something is clearly not enough to make an automation project a success and some of the other important things to take into account are listed in Table 1.3. These issues are excluded from the scope of this thesis but they are listed here for completeness sake. They also help to understand how many issues there are to handle when conducting test automation projects and how big and challenging these projects can be.

When to automate.	Tool selection process.
What to automate.	Taking tools and frameworks into use.
What to automate first.	Test planning and design.
What can be automated.	Effect to development and testing processes.
How much to automate.	Other organizational issues.
Build or buy.	Costs and savings.
Build in-house or use consultants.	Metrics.

Table 1.3: Some important issues excluded from the scope

1.4 Methodology

Hevner et al. (2004) assert that software engineering research is characterized by two complementary paradigms: behavioral science and design science. Behavioral science is theoretical and has its origins in natural science while design science originates from engineering and tries to find practical solutions for research problems from the field. The importance of practical research and concrete, evaluated results is emphasized both by Hevner et al. (2004) and Shaw (2002) and the methodology of this thesis is based on models presented by them.

Much of the knowledge of this thesis comes from multiple testing and test automation books and articles listed in the bibliography. This is not only a literature study, however, as I have six years experience from testing and test automation using different tools and techniques. This experience gives first hand knowledge about the subject and also greatly helps processing the information read from the literature.

1.5 Goals

The objective of the thesis is presenting a concept for a large scale test automation framework. To achieve this goal I have to go through the following four steps which are also requirements for this thesis.

1. Define requirements for large scale test automation frameworks.
2. Design a framework meeting these requirements.
3. Test the designed framework against the defined requirements in a pilot.
4. Collect results from the pilot and validate the feasibility of the framework based on them. Adapt requirements based on the new information if necessary.

1.6 Structure

This chapter has presented promises and problems of test automation as a motivation for the thesis and different automation approaches as background information. Also scope, methodology and requirements for the thesis have been defined.

Chapter 2 defines requirements for successful large scale test automation frameworks and discusses ways how to meet these requirements.

Chapter 3 is the core of the thesis. In that chapter a framework meeting the requirements set in Chapter 2 is introduced and explained in detail.

In Chapter 4 the presented framework is tested against defined requirements. This is done by implementing a prototype of the framework and using it in two simple but non-trivial pilots testing a Windows application and a web page.

Results and experiences from the pilot are collected together in Chapter 5. Based on the results the general feasibility of the suggested framework is evaluated and possibly changes made to the requirement set defined earlier.

Chapter 6 is left for conclusions. There it is evaluated how well requirements for this thesis, defined in Section 1.5, are met.

Chapter 2

Requirements for Test Automation Frameworks

Chapter 1 introduced different views to test automation and stated that the scope of this thesis is large scale test automation frameworks. The aim of the chapter is to define requirements for these frameworks and discuss methodologies and techniques needed to meet the requirements.

2.1 High Level Requirements

A short list of high level requirements applicable for all large scale test automation frameworks, derived from Kit (1999) and other sources, is shown in Table 2.1.

Automatic test execution is a high level functional requirement for test automation frameworks. How it can be accomplished and how it breaks into more detailed requirements is discussed in Section 2.2. Ease of use and maintainability are non-functional in their nature and in general achieving them is harder than simply implementing the functional requirements. Sections 2.3–2.8 present methods and techniques how they can be fulfilled and how they turn into more detailed requirements. Detailed requirements are finally summarized in Section 2.9.

2.2 Framework Capabilities

This section explains what capabilities a test automation framework must have. Discussed features can be seen as functional requirements for automation frameworks.

Automatic Test Execution	Fully automatic test execution is of course the number one requirement for test automation frameworks. Just executing tests is not enough, however, and the framework must also be capable to for example analyze test outcome, handler errors and report results. (Kit, 1999)
Ease of Use	Framework must be easy to use by test engineers or it is very likely to be abandoned. Framework users must be able to design and edit tests, run them and monitor their status easily without any programming skills. (Kaner, 1997; Kit, 1999; Nagle, 2000)
Maintainability	It must be easy and fast to maintain both test data and framework code when the tested system changes or updates are needed otherwise. It should also be easy to add new features to the framework. (Fewster and Graham, 1999; Kit, 1999; Pettichord, 1999; Nagle, 2000; Zallar, 2001; Rice, 2003)

Table 2.1: High level requirements for test automation frameworks

2.2.1 Executing Tests Unattended

Framework must be able to start executing tests with a push of a button and run tests on its own. This means that framework must be able to set up the test environment and preferably also check that all preconditions are met. (Fewster and Graham, 1999; Pettichord, 1999)

2.2.2 Starting and Stopping Test Execution

It must be possible to start test execution manually. It is also convenient if tests can be started automatically at a specified time or after a certain event (e.g. new version of the SUT available). Easiest way to start execution at a certain time is making it possible to start test execution manually from command line and using operating system's features for scheduling (at in Windows and cron in Unixes). Starting after predefined events can be implemented similarly using external tools.

2.2.3 Handling Errors

Part of running tests unattended is recovering from errors caused by the tested system or the test environment not working as expected. Test framework ought to notice error situations and continue testing without manual intervention. Handling all possible but rare errors is seldom worth the effort, though, and over-engineering should be avoided. (Pettichord, 1999)

2.2.4 Verifying Test Results

An integral part of test execution is verifying test results. Fewster and Graham (1999) define verification as one or more comparisons between *actual outcome* of a test and predefined *expected outcome.* They also explain different comparison techniques which are out of the scope of this thesis.

2.2.5 Assigning Test Status

After a test is executed and its results verified it ought to be given a status. If the test was executed without any problems and all comparisons between actual and expected outcomes match the test gets a pass status. In every other case the status is fail. Besides the status every test case should also get a short but descriptive status message. For passed tests this message is normally not that important but with failed tests it can give details about the cause of the problem (e.g. "Calculation failed: expected 3, got 2" or "Initializing test environment failed: Not enough space on disk").

Some authors, e.g. Fewster and Graham (1999), propose having more statuses than pass and fail. Other statuses could for example differentiate expected failures from new ones and problems caused by test environment from other problems. While I understand the motivation I suggest using the status message for those purposes instead.

2.2.6 Handling Expected Failures

The most frustrating part of failure analysis is going through test cases which are known to fail and checking whether they have failed similarly as before or have new defects appeared. That is wasteful and clearly something that should be automated as Fewster and Graham (1999) recommends.

To be able to differentiate *expected failures* from new ones the framework must know what is the expected outcome when test fails. This means that the framework must store both the expected outcome and the expected failed outcome of the test. When a test fails, and is expected to do so, the expected failed outcome can be

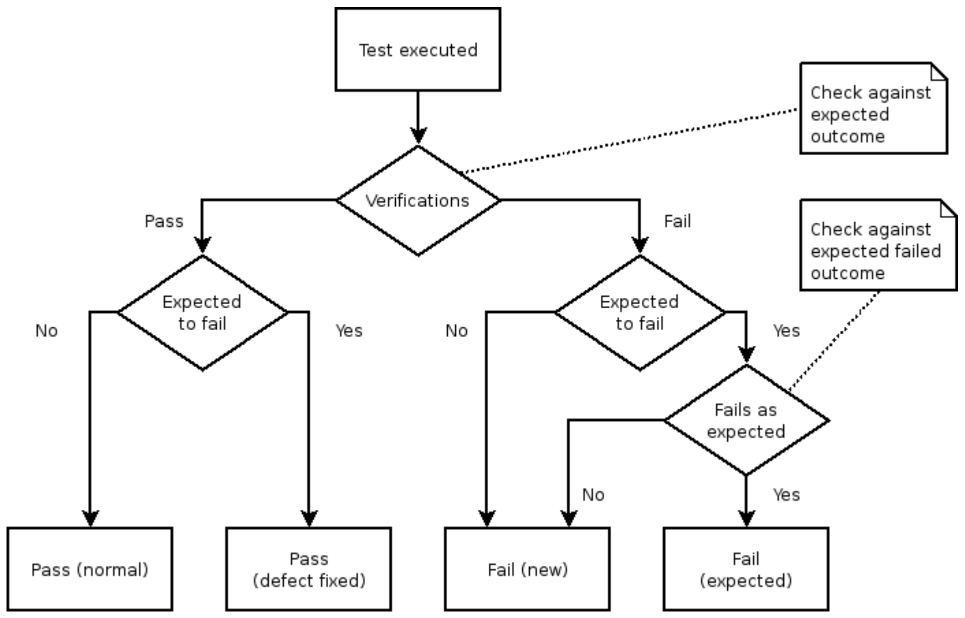


Figure 2.1: Handling expected failures

compared against the actual outcome as seen in the Figure 2.1. If outcomes match an expected failure is detected, otherwise a new defect has been found. I recommend communicating expected failures using status messages but, as already discussed, it can be argued that a specific test status should be used instead. In either case it sure is fast to analyze failed test which right away tells something like “Expected failure. Defect id: #2712”.

Fewster and Graham (1999) also recommend notifying test engineers when a test which is expected to fail passes (i.e. defect has been fixed). This way they can manually verify that the test really has passed and defect is fixed without side effects. After every passed test the framework should check whether the test was expected to fail by checking does it have the expected fail outcome. If test was expected to fail it should be labeled accordingly—I recommend setting the status message to something like “Defect #2712 fixed”. The whole process of handling expected failures is illustrated by a flowchart in Figure 2.1.

2.2.7 Detailed Logging

Test automation framework ought to give enough information to test engineers and developers so that they can investigate failed test cases. Giving only test statuses and short status messages is not enough. Instead more detailed test logs about the

Fail	Used to log the status of a failed test case.
Pass	Used to log the status of a passed test case.
Fatal	Used when an unrecoverable failure preventing further test execution occurs. This kind of problem is most often in the test environment (e.g. disk full or network down).
Error	Used when an unexpected error preventing current test's execution occurs. For example the button to push in a test does not exists.
Failure	Used when the test outcome does not match the expected outcome.
Warning	Used when a recoverable error not affecting test execution occurs. For example deleting some temporary file fails.
Info	Used to log normal test execution like starting test and passed verifications.
Debug	Gives detailed information about what the test automation framework or a particular test case is doing. Used mainly for debugging the framework or test cases.
Trace	Similar to debug but contains even more detailed information.

Table 2.2: Suggested logging levels

test execution process and how the tested system behaved are needed. On top of that the framework must log what it is doing internally to make it easier to debug problems in the framework itself.

The main dilemma with any kind of logging, as stated by Pettichord (2002), is how much to log. Logging too much makes finding relevant information hard and huge log files cause a storage problem. Then again logging too little can make logs totally useless. There is no perfect solution for this problem but for example using different logging levels make the problem smaller.

Multiple logging levels provide a mechanism for controlling how much detail is captured. For example lowest levels, those that produce most output, can in normal cases be filtered out before they are even logged. If detailed information is needed the filter can be adjusted. Later when logs are viewed entries can be filtered even further so that only interesting levels are viewed. What levels are needed depends ultimately on the context but I have found the list in Table 2.2 pretty good for most situations. Two examples how these levels can be used are found in Figure 2.2.

Timestamp	**Level**	**Message**
20050502 15:14:01	Info	Test execution started. SUT: Calculator v. 0.6.1, Test suite: Calculator basic
20050502 15:14:04	Info	Test environment setup without problems
20050502 15:14:04	Info	Starting test case ’Add 01’
20050502 15:14:05	Info	Test ’1 + 2’ returned 3
20050502 15:14:05	Pass	’Add 01’
20050502 15:14:06	Info	Starting test case ’Add 02’
20050502 15:14:06	Info	Test ’0 + 0’ returned 0
20050502 15:14:06	Pass	’Add 02’
20050502 15:14:07	Info	Starting test case ’Add 03’
20050502 15:14:08	Failure	Test ’1 + -1’ returned 2, expected 0
20050502 15:14:08	Fail	’Add 03’ failed: ’1 + -1’ returned 2, expected 0
20050502 15:14:09	Info	Starting test case ’Add 04’
20050502 15:14:09	Info	Test ’0.5 + 0.5’ returned 1
20050502 15:14:09	Pass	’Add 04’
20050502 15:14:10	Info	Test execution ended. 4/4 test cases executed (100 %), 3/4 passed (75 %), 1/4 failed (25 %).

Timestamp	**Level**	**Message**
20050502 16:23:31	Info	Test execution started. SUT: Calculator v. 0.6.2, Test suite: Calculator basic
20050502 16:23:43	Fatal	Setting up test environment failed
20050502 16:23:43	Info	Test execution ended. 0/4 test cases executed (0 %), 0/4 passed (0 %), 0/4 failed (0 %).

Figure 2.2: Two test log examples

As already mentioned the level used for filtering log entries at the execution time (often called simply logging level) must be configurable (Pettichord, 2002). Setting a log level somewhere means that entries lower than the threshold level are not to be logged. Adjusting the level must be easy for anyone executing tests and it should preferably be possible also while tests are running. The default logging level should normally be info. Info level gives required information about test execution but no debugging details which should not be needed normally. For example logs in Figure 2.2 would be too uninformative if higher level was used and unnecessarily long if level was lower.

2.2.8 Automatic Reporting

Test logs have all the information from test execution but, especially if they are long, they are not good for seeing test statuses at a glance. This kind of view is provided by concise test reports. Test reports provide statistical information about the quality of the tested system and they are not only important for test engineers but also for developers, managers and basically everyone involved with the project. (Fewster and Graham, 1999)

Reports should not be too long or detailed. Having a short summary and list of executed test cases with their statuses is probably enough. List of items every test report should have is presented below—adapted from Buwalda et al. (2002)—and Figure 2.3 shows a simple example.

- Name and version of the tested system.
- Identification of the executed test suite.
- The total number of passed tests and executed tests.
- List of errors found.
- Optionally a list of all executed tests along with their statuses.

Test reports can be either created at the same time when tests are run or they can be constructed based on test logs afterwards. In my experience the latter method is easier because then you only need to think about logging, not reporting, while writing test code. It also makes reporting flexible because if more information is needed later it can be parsed from logs and test code does not need to be changed. Example report in Figure 2.3 is constructed based on earlier example test log in Figure 2.2.

After tests are executed the created report should be published so that results are easily available for everyone interested. Publishing can mean for example sending report to some mailing list or uploading it to a web page. Published results should also have a link or a reference to previous test results and to more detailed test logs.

SUT	Calculator v. 0.6.1	
Test suite	Calculator basic	
Start time	20050502 15:14:01	
End time	20050502 15:14:10	
Tests executed	4/4 (100 %)	
Tests passed	3/4 (75 %)	
Tests failed	1/4 (25 %)	
Failed Tests		
20050502 15:14:08	Fail	'Add 03' failed: '1 + -1' returned 2, expected 0
All Executed Tests		
20050502 15:14:05	Pass	'Add 01'
20050502 15:14:06	Pass	'Add 02'
20050502 15:14:08	Fail	'Add 03' failed: '1 + -1' returned 2, expected 0
20050502 15:14:09	Pass	'Add 04'

Figure 2.3: Example test report

2.3 Modularity

This section discusses the importance of modularity in test automation framework design.

2.3.1 Linear Test Scripts

Linear test scripts do not use any external functions and, as illustrated in Figure 2.4, they interact directly with the tested system. Simple linear scripts are fast to write initially and they are thus well suited for small tasks. The problem with linear scripts is that when tasks get bigger and more complicated also they grow longer, more complicated and generally hard to maintain. They also tend to have repeating code which could be taken out and reused. If many similar test scripts are created the maintenance problem just grows bigger because one change in the tested system can require changes in every script. (Fewster and Graham, 1999)

2.3.2 Test Libraries and Driver Scripts

Structured coding is a common programming paradigm and it is of course applicable with test code too (Kelly, 2003). In low level it can mean simply using small functions in the same file as the main test script. In higher level use test functions are placed to external, shared *test libraries* from where they can be used by any test script.

When test libraries do most of the actual testing work test scripts can be short and simple. Because scripts are only driving the test execution they are customarily called *driver scripts*. Figure 2.5 shows how two driver scripts uses same test library to interact with the tested system.

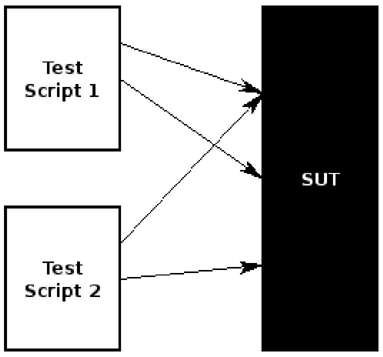


Figure 2.4: Linear test scripts

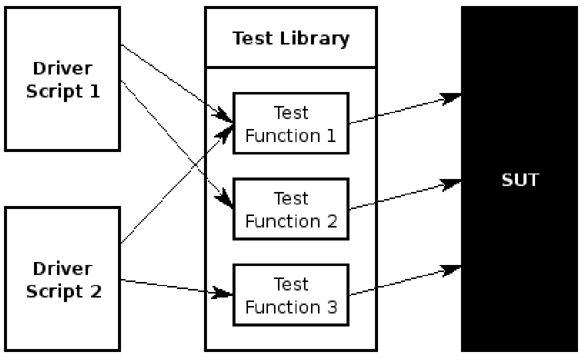


Figure 2.5: Driver scripts and a test library

2.3.3 Promises and Problems

When enough functionality is available in easy to use test libraries creating new driver scripts for different tests is very easy and fast. This is very efficient code reuse but an even bigger advantage of this approach is that it makes maintenance simpler. When something is changed in the tested system updating only one function in the test library can make everything work again and even in worst cases changes should be needed only in the test library. (Zambelich, 1998; Fewster and Graham, 1999; Kelly, 2003; Rice, 2003)

The problem with test libraries is that creating them is not easy. Fewster and Graham (1999) mention that keeping track of, documenting, naming, and storing of created scripts is a big task. They also note that if reusable functions are not easy to find and well documented automators are likely to write their own versions. Marick (1997) notes that writing well-designed API to a test library is as hard as writing any good API and it should not be expected to be right the first time. Similar problems have been noticed also by Kaner et al. (2001) and Pettichord (1999) who are pretty skeptical about test libraries in general. They remark that effort required to build a useful library is not always justified.

In my opinion modularity and test libraries are a must when building large test automation frameworks. I cannot see how maintainability could be achieved without them. Libraries sure are an overkill for simple cases but in those cases the whole idea of building a big framework is questionable.

2.4 Data-Driven Testing

2.4.1 Introduction

Simple test scripts have test data embedded into them. This leads to a problem that when test data needs to be updated actual script code must be changed. This might not be a big deal for the person who originally implemented the script but for a test engineer not having much programming experience the task is not so easy. If the script is long and non-structured the task is hard for everyone. Another problem with having the test data inside test scripts is that creating similar tests with slightly different test data always requires programming. The task may be easy—original script can be copied and test data edited—but at least some programming knowledge is still required. This kind of reuse is also problematic because one particular change in the tested system may require updating all scripts. (Strang, 1996; Nagle, 2000)

Because of these problems embedding test data into scripts is clearly not a viable solution when building larger test automation frameworks. A better approach is reading the test data from external data sources and executing test based on it. This approach is called data-driven testing and it is illustrated in Figure 2.6. External test data must be easily editable by test engineers without any programming skills. It is often in tabular format and edited in spreadsheet programs, as seen in Figure 2.7. (Strang, 1996; Fewster and Graham, 1999; Nagle, 2000; Kaner et al., 2001; Rice, 2003; Mugridge and Cunningham, 2005)

2.4.2 Editing and Storing Test Data

Because data-driven test data is tabular it is natural to use spreadsheet programs to edit it. Test engineers and business people are, as Mugridge and Cunningham (2005) point out, also likely to be familiar with spreadsheet programs and already have them available. Another benefit is that spreadsheet programs are often used for simple test management tasks and in that case there is less need for storing same data in different places or copying it back-and-forth.

Spreadsheet files can be saved in comma-separated-values (CSV), tab-separated-values (TSV) or spreadsheet programs' native formats. CSV and TSV files are handy because they are very easy to parse but unfortunately most spreadsheet programs seem to alter the data whey they open these files. For example if I store phone number +358912345 and version number 1.5.1 to a CSV file and open it to my Excel they are "auto-corrected" to 358912345 and 1.5.2001, respectively. The easiest solution for this problem is storing data using program's native format and only exporting it into CSV or TSV. Another possibility is processing the native format directly but it requires some more initial effort.

HTML tables provided another easy-to-parse format for presenting test data. Editing HTML with a decent graphical editor is nearly as convenient as using a spreadsheet program but unlike their native formats HTML can also be edited with any text editor if nothing else is available.

Storing test data into any kind of flat file has scalability problems in very large scale use. If for example test data is created and edited in several workstations and used in multiple test environments it gets hard to have the same version of the data everywhere. Configuration management and clear instructions surely help but even they may not be enough. The real solution for scalability problems is storing

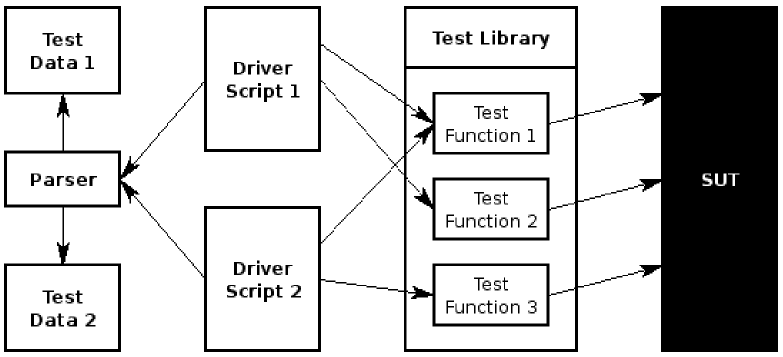
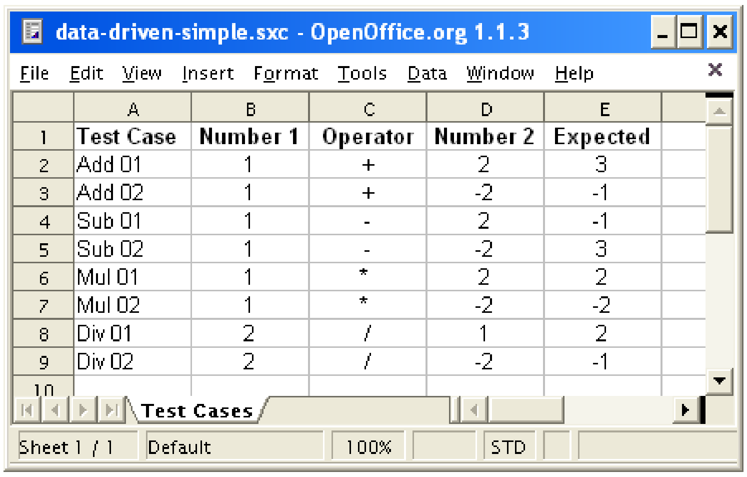


Figure 2.6: Data-driven approach


data-driven-simple.sxc - OpenOffice.org 1.1.3
File Edit View Insert Format Tools Data Window Help
Test Cases
Sheet 1 / 1
Default
100%
STD

	A	B	C	D	E
1	Test Case	Number 1	Operator	Number 2	Expected
2	Add 01	1	+	2	3
3	Add 02	1	+	-2	-1
4	Sub 01	1	-	2	-1
5	Sub 02	1	-	-2	3
6	Mul 01	1	*	2	2
7	Mul 02	1	*	-2	-2
8	Div 01	2	/	1	2
9	Div 02	2	/	-2	-1
10					

Figure 2.7: Data-driven test data file

```
data = open('testdata.tsv').read()
lines = data.splitlines()[1:]   # [1:] excludes the header row

for line in lines:
    testId, number1, operator, number2, expected = line.split('\t')
    # Actual test functionality excluded
```

Listing 2.1: Simple data-driven parser

test data into a central database. That way the test data can be edited using for example a web-based interface and driver scripts can query the data directly from the database. Building such a system is a big project itself and can be recommended only if scalability is a real problem.

Bird and Sermon (2001) and Bagnasco et al. (2002) have had good experience using XML for storing the test data. I agree that XML has some good characteristics but because using it would require implementing a separate test design system it is not a good solution for situations where spreadsheets and HTML editors are sufficient. For very large scale use I would then again rather select a database with a web-based user interface because it would also provide a centralized data storage.

2.4.3 Processing Test Data

Implementing a script for parsing data-driven test data can be surprisingly easy with modern scripting languages. For example data in Figure 2.7 can be exported into a TSV file and parsed with four lines of Python code in Listing 2.1. In the example test data is first read into `data` variable and then split to lines so that header row is excluded. Data lines are then processed one by one. Lines are split to cells from tabulator character and cells' data assigned to variables making test data available in the script for actual testing (which is excluded from the example).

In Listing 2.1 the parsing logic is in the same script as the actual testing code which is against modularity goals presented in Section 2.3. This simple parser is also limited in functionality and would fail for example if data had any empty rows or columns were rearranged. If more functionality is required from the parser it should be implemented as a common module which all driver scripts can use—this was already seen in the Figure 2.6. How the parser should be implemented and what kind of functionality it needs is further discussed in Section 3.2 and Chapter 3 in general.

2.4.4 Promises and Problems

Main benefit of data-driven test automation is that it makes creating and running lots of test variants very easy (Strang, 1996; Fewster and Graham, 1999; Kaner et al., 2001). Editing tests and adding new similar ones is easy and requires no programming skills (Kaner, 1997). Another benefit is that test data can be designed and created before test implementation or even before the tested system is ready

(Strang, 1996; Pettichord, 2003) and it is usable in manual testing even if automation implementation is never finished. Overall, the data-driven approach is the key for ease-of-use in large scale test automation.

The data-driven approach also helps with maintenance. When the tested system changes it is often possible to change only either the test data or the test code and their maintenance responsibilities can also be divided between different people. (Strang, 1996; Kaner, 1997; Marick, 1997; Kaner et al., 2001; Rice, 2003)

The biggest limitation of the data-driven approach is that all test cases are similar and creating new kinds of tests requires implementing new driver scripts that understand different test data. For example test data in Figure 2.7 and its parsing script in Listing 2.1 are designed for testing calculations with only two numbers and would require major changes to handle longer tests like $5 * 8 + 2 = 42$. In general test data and driver scripts are strongly coupled and need to be synchronized if either changes. Another disadvantage of data-driven testing is the initial set-up effort which requires programming skills and management. (Fewster and Graham, 1999)

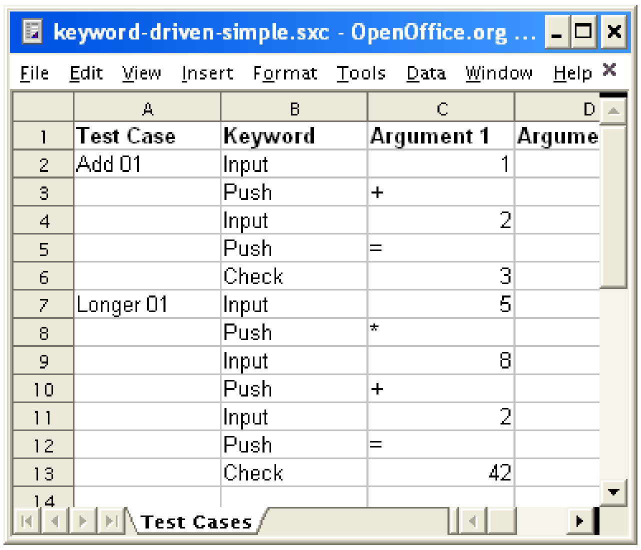
2.5 Keyword-Driven Testing

2.5.1 Introduction

Previous section introduced data-driven testing and stated that it has multiple promises. It also mentioned that its biggest limitation is that all test cases are similar and creating totally new tests requires programming effort. A solution for this limitation, offered by Fewster and Graham (1999) and Kaner et al. (2001) among others, is the keyword-driven approach where not only the test data but also directives telling what to do with the data are taken from test scripts and put into external input files. These directives are called *keywords* and test engineers can use them to construct test cases freely. The basic idea—reading test data from external files and running tests based on it—stays the same as in data-driven testing. As Fewster and Graham (1999) put it, keyword-driven testing is a logical extension to data-driven testing. Example of a keyword-driven test data file is seen in Figure 2.8.

2.5.2 Editing and Storing Test Data

There is no real difference between handling keyword-driven and data-driven test data. In both cases spreadsheets or HTML tables are normally adequate but a central database can solve scalability problems in very large scale use.


keyword-driven-simple.sxc - OpenOffice.org ...
File Edit View Insert Format Tools Data Window Help
Test Cases

	A	B	C	D
1	Test Case	Keyword	Argument 1	Argume
2	Add 01	Input	1	
3		Push	+	
4		Input	2	
5		Push	=	
6		Check	3	
7	Longer 01	Input	5	
8		Push	*	
9		Input	8	
10		Push	+	
11		Input	2	
12		Push	=	
13		Check	42	
14				

Figure 2.8: Keyword-driven test data file

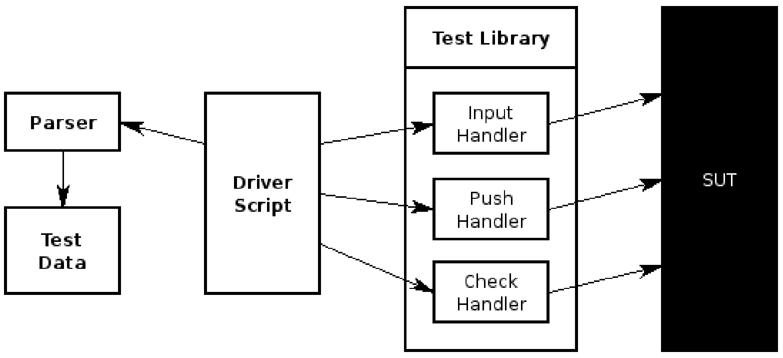


Figure 2.9: Handlers for keywords in Figure 2.8

2.5.3 Processing Test Data

Processing a simple keyword-driven test data file as in Figure 2.8 is not much different from processing a simple data-driven test data file. Simple parser, much like the one in Listing 2.1, can get keywords and their arguments to the driver script. In more complex cases the parser should again be implemented only once and placed into a common, reusable module.

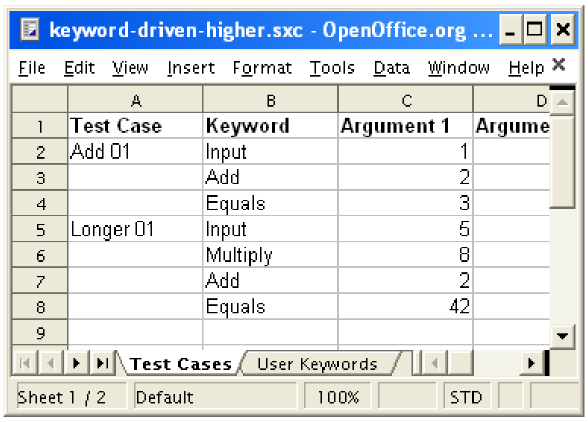
After the test data is parsed the driver script must be able to interpret keywords and execute the specified action using assigned arguments (Fewster and Graham, 1999). To keep the framework modular it is a good idea to have a handler (i.e. a test function) for each keyword and put these handlers to external test library as presented in Figure 2.9.

2.5.4 Keywords in Different Levels

One of the big decisions to make when designing a keyword-driven framework is the level of the keywords to be used. In Figure 2.8 keywords are pretty low level (e.g. Input, Push) making them suitable for detailed testing on the interface level. When testing higher level functionality like business logic low level keywords tend to make test cases very long and higher level keywords (e.g. Add in our calculator example) are much more usable. Sometimes it is possible to use only either low or high level keywords and build keyword handlers accordingly. Often both levels are needed and then it is a good idea to construct higher level keywords from low level keywords. For example low level keywords Input and Push could be used to create higher level keywords Add, Subtract, Multiply and Divide and similarly Equals could be created using Push and Check. Figure 2.10 shows how these higher level keywords make test cases shorter.

A straightforward way to construct new high level keywords is letting framework developers implement handlers for them in the framework so that new handlers use lower level handlers as seen in Figure 2.11. Implementing higher level keywords directly into the framework has its limitations, though, as test engineers are limited only to keywords in the test library and they cannot easily create new ones.

A more flexible solution is allowing test engineers to construct higher level keywords using the same interface they use for designing test cases. These kind of techniques are presented with different implementation both by Buwalda et al. (2002) and Nagle (2000). Figure 2.12 shows one option how new keywords could be constructed. To


keyword-driven-higher.sxc - OpenOffice.org ...
File Edit View Insert Format Tools Data Window Help
Test Cases
User Keywords
Sheet 1 / 2
Default
100%
STD

	A	B	C	D
1	Test Case	Keyword	Argument 1	Argume
2	Add 01	Input	1	
3		Add	2	
4		Equals	3	
5	Longer 01	Input	5	
6		Multiply	8	
7		Add	2	
8		Equals	42	
9				

Figure 2.10: Keyword-driven test data file with higher level keywords

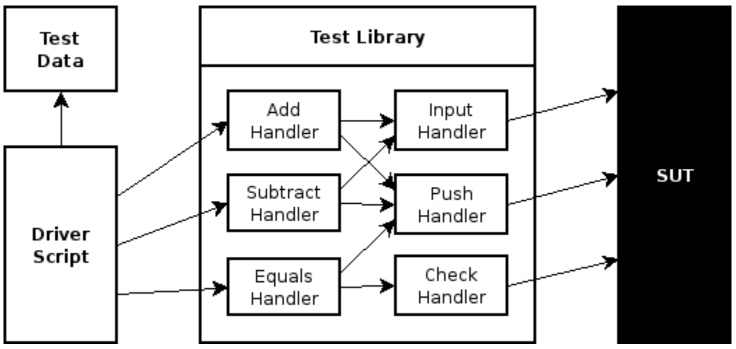
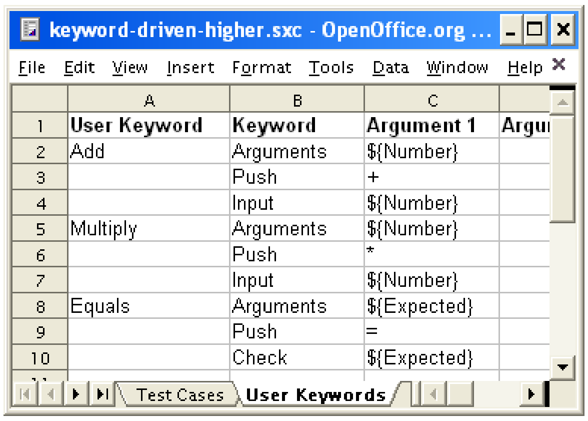


Figure 2.11: Creating higher level keywords in test library


keyword-driven-higher.sxc - OpenOffice.org ...
File Edit View Insert Format Tools Data Window Help
Test Cases
User Keywords

	A	B	C	
1	User Keyword	Keyword	Argument 1	Argu
2	Add	Arguments	${Number}	
3		Push	+	
4		Input	${Number}	
5	Multiply	Arguments	${Number}	
6		Push	*	
7		Input	${Number}	
8	Equals	Arguments	${Expected}	
9		Push	=	
10		Check	${Expected}	

Figure 2.12: Creating higher level keywords in test design system

differentiate these new keywords from those implemented directly in the test library I like to call them *user keywords* and *base keywords*, respectively. Test libraries do not have handlers for user keywords, instead they are executed by executing base keywords, or other user keywords, they are constructed from.

The main benefit in making it possible to construct new keywords using the test design system is that new keywords can be created and maintained easily without programming skills. Of course it can be argued that creating user keywords is already programming but at least it is so easy that everyone involved with test automation can learn it. The biggest problem with the approach is that processing test data gets more complicated but, once again, the small increase in initial effort is insignificant when building a large scale automation framework. Implementation issues are discussed in greater detail later in Chapter 3 and especially in Section 3.3.

2.5.5 Promises and Problems

Keyword-driven testing has all the same benefits as data-driven testing. As already mentioned the main advantage over data-driven testing is that the keyword-driven technique makes it easy even for non-programmers to create new kinds of tests. Comparing Figures 2.7 and 2.8 prove this point pretty explicitly. Keyword-driven testing is a big step forward from pure data-driven testing where all tests are similar and creating new tests requires new code in the framework.

One problem with the keyword-driven approach is that test cases tend to get longer and more complex than when using the data-driven approach. For example test case Add 01 is only one line in Figure 2.7 but five lines in Figure 2.8. This problem is due to greater flexibility and can be pretty well solved using higher level keywords. The main problem of keyword-driven testing is that it requires a more complicated framework than data-driven testing.

2.6 Other Implementation Issues

Same processes, methodologies and techniques which are important when designing and implementing any non-trivial software system are important also when building larger test automation frameworks (Zallar, 2001). Those implementation issues which I feel are the most important are discussed in this section.

2.6.1 Implementation Language

Programming languages are needed when building test automation tools and frameworks. Often automation tools are also controlled using input scripts which have their own language. Requirements for language used when building a tool and for language used with the tool can be very different. With multiple programming languages available picking the right choice in different situations is not an easy task. Selecting a language already used in the organization or known otherwise is always a safe bet but it is not the only criteria.

System Programming Languages

System programming languages such as C++ and Java are designed for building data structures, algorithms and large systems from scratch (Ousterhout, 1998). They are optimized for performance and scalability rather than for fast programming speed. These languages may well be used when building test automation tools and frameworks especially when developers know them well already and do not know suitable higher level scripting languages. For a language to be used as an input script for a tool they are too low level.

Scripting Languages

Scripting languages such as Perl, Python, Ruby, Tcl and Visual Basic are higher level languages than system programming languages. Ousterhout (1998) states that instead of building things from scratch they are designed for gluing components and systems together. Ousterhout (1998) suggests that programming something takes five to ten times less code in scripting language than in system programming language. Because programmers write approximately the same number of lines in same time regardless the language, scripting languages may fasten development time with the same magnitude. Shorter programs are also easier to maintain.

Ousterhout (1998) mentions that scripting languages are somewhat slower than system programming languages, but current computers are so fast that it is normally not important. Pettichord (1999) notes that in test automation this slight performance difference is hardly ever a problem because most of the time is spent waiting for the tested system to respond for inputs or setting up test environment.

I have experience using both system programming and scripting languages for test automation and I can second all the positive things Ousterhout (1998) says about the latter. They are excellent for implementing tools and frameworks and also as input scripts used by tools.

Shell Scripts

Shell scripts are lower level scripting languages which come with the operating system. In Unixes the primary shell scripting language is Bash and Windows machines natively have primitive DOS batch files. Shell scripts are not as flexible as higher level scripting languages but they are very handy for small tasks. I use shell scripts regularly for small automation tasks but I consider them too cumbersome for implementing larger frameworks.

Vendor Scripts

Vendor scripts are proprietary languages implemented by tool vendors to be used with these tools. Test tools using vendor script should be approached with a certain caution for reasons listed below.

- Nobody knows them. Developers and some test engineers are likely to know many programming languages but not likely to know some special vendor script. Hiring people with previous knowledge is hard. (Pettichord, 2001)
- Not mature enough. Vendor scripts do not have those features mature programming languages and their support libraries have. If vendor script does not have for example good enough string handling capabilities you have to implement them yourself. (Pettichord, 2001)
- Vendor scripts have compatibility problems with other tools and languages. (Rice, 2003)

Because of these reasons my advice is avoiding vendor scripts altogether if possible. Luckily they are getting rare nowadays as vendors have started embedding existing mature scripting languages to their products. Examples of this are Mercury's QuickTest Pro which uses Visual Basic and PushToTest's TestMaker using Jython (a Java implementation of the Python language).

2.6.2 Implementation Technique

Manual Programming

Manual script writing is a normal programming task and if done properly scripts are easy to use and maintain. It has two obvious drawbacks: writing scripts takes some time and requires programming skills which test engineers may not have. (Fewster and Graham, 1999; Mosley and Posey, 2002)

Capture and Replay

Capture and replay tools are supposed to overcome problems with manual script writing. The idea is that you push a record button and let the tool record every action—keystroke, mouse movement or click—while executing test manually. The tool stores captured actions in a script for later playback. This approach sounds attractive and, as Meszaros (2003) note, it may be suitable for situations where scripts are needed fast, they can be trashed in the future and programming skills required for manual scripting are not available.

The biggest problem with capture and replay is that maintaining recorded scripts is nearly impossible. More problems associated with it are listed for example by Kaner (1997), Zambelich (1998), Pettichord (1999), Kaner et al. (2001), Zallar (2001), Mosley and Posey (2002), Meszaros (2003), and Rice (2003). Because of these numerous problems I do not consider capture and replay an option when building larger test automation frameworks.

2.6.3 Testware Architecture

Test automation frameworks require plenty different artifacts from scripts to test data and from documentation to test results. All these artifacts are commonly called *testware*. Testware architecture is about managing the testware in an order to make the framework easy to use and maintain. The bigger the framework is the greater the need for detailed testware architecture. (Fewster and Graham, 1999)

Testware architecture is important both in manual and automated testing. Discussing it thoroughly is out of the scope of this thesis but this section goes through few aspects I feel most important when talking about larger test automation frameworks.

Version Control

Version control and configuration management are required for any larger software project and test automation frameworks are no exception. Maintaining all the needed testware and its different versions is a nightmare without an adequate version control system. (Fewster and Graham, 1999; Zallar, 2001; Buwalda et al., 2002)

Coding and Naming Conventions

Coding and naming conventions cover everything from variable, function and file naming to code indentation. The idea of coding conventions is that when everyone writes similar code it is then easy to understand and maintain by anyone. Consistent naming of modules, functions and files greatly helps with reuse when module and function names are easy to remember and even guess. (Fewster and Graham, 1999)

These kinds of convention are very important when anything larger is programmed and should not be forgotten when building large test automation systems. Conventions should be agreed and documented when starting a test automation project. Often this is easy as conventions used when implementing tested systems can be used straight away or with slight modifications.

Documentation

Lack of adequate documentation makes using test libraries hard because people new to them need to spent time finding how and under what circumstances to use them. Documentation is also needed in maintenance. Need for quality documentation is commonly understood but there are two problems: first the documentation must be written and then it must be kept updated. (Fewster and Graham, 1999)

For code documentation purposes semi-automatic documentation systems such as Javadoc are getting more and more popular. With these tools the original documentation is kept within the code so that it is available when code is read and edited and it can easily be kept updated with the code. Documentation can be extracted with special tools so that it is readable with a web browser or some other viewer. Tools can also read function and variable names from the code so that they do not need to be documented manually.

Code documentation is very important but not enough on its own. Other needed documents include instructions for setting up, using and maintaining the framework. It is the quality of the documentation, not its quantity, that is important. (Fewster and Graham, 1999)

2.7 Testability

Most often the hardest part of a test automation effort is integrating the selected tool with the tested system. Pettichord (2002) and Kaner et al. (2001) state that instead of fine tuning the tool to handle all possible quirks of the tested system it might be better to invest on increasing the *testability* of the system. Pettichord (2002) has noticed that the success of the whole automation project may well depend on testability issues and also others, e.g. Fewster and Graham (1999), Kaner et al. (2001) and Bach (2003), underline importance of testability. Testability problems are similar in small and large scale test automation projects but in larger scale its positive effect to maintenance gets more and more important.

Pettichord (2002) defines testability as visibility and control and says that testability means having reliable and convenient interfaces to drive the test execution and verification. This definition maps very well to the two steps of functional testing in Figure 1.1: we need control when doing something to the tested system and visibility when verifying the test outcome. Pettichord (2002) states that the key for testability is close cooperation between test engineers and programmers. Otherwise test engineers may be afraid to ask for testability improvements they need and programmers are unaware of what features are needed. Fewster and Graham (1999) remark that what needs to be tested and how it can be tested should be decided while the system is being designed and implemented and not afterwards.

2.7.1 Control

Software testing occurs through an interface and this interface has a huge impact to the testability and especially its control part. Interfaces, as discussed in Section 1.2.3, come in different levels and are meant either for humans or systems. All kinds of programming interfaces inherently have high testability because test tools can interact with them similarly as other programs. Interfaces used by humans can be either graphical or non-graphical, and non-graphical ones can be further divided into textual interfaces and command line interfaces (CLI). Command line interfaces normally have very high testability and they can be driven even with simple shell scripts. Textual interfaces are not that trivial but simple tools (e.g. Expect in Unixes) can be easily used to control them. (Pettichord, 1999, 2002)

Graphical user interfaces (GUI) are not designed for automation (Marick, 2002) and are technically complicated (Kaner et al., 2001). This leads to low testability and it is no wonder that problems with GUI test automation are reported by several authors like Pettichord (1999, 2002), Kaner et al. (2001), Marick (2002) and Bach (2003). In practice GUI automation is complicated because user interface objects (often called widgets) are hard to recognize and interact with automatically. Special tools are needed and even they may not be able to work with non-standard or custom controls (Pettichord, 2002). GUIs also tend to change frequently (Marick, 1997, 2002; Pettichord, 2002) and in worst cases changes occur at the very end of the project when system functionality ought to be fully regression tested. Human tester can find a button which has changed its text from Save to Submit and moved to different position but that is not so easy for computers.

The number one solution for problems associated with GUI test automation is trying to avoid it altogether (Kaner et al., 2001; Marick, 2002; Pettichord, 2002). System functionality is tested much more easily below the GUI using an interface which does not change as often and where test tools can be hooked without a hassle. If the well known development practice to separate *presentation code* and *domain code* is obeyed testing below the GUI should be easy (Fowler, 2001; Pettichord, 2002). Sometimes there is a need to add a new testing interface to give even more control to testing. As Pettichord (2002) and Marick (1997) note some people think that mixing test code with product code undermines the integrity of the testing but neither they nor Fowler (2001) share that feeling. Fowler (2001) even asserts that testability is an important criteria for good design and notes that hard-to-test applications are very difficult to modify.

Testing below GUI of course leaves the user interface untested but verifying that GUI looks and works correctly is anyway something where humans are better than computers. Humans can easily tell if everything is not correct while computers find problems only from places where they are told to look them. Humans also notice non-functional issues like usability problems and missing features. (Marick, 2002)

There are of course valid reasons for GUI test automation. Pettichord (1999) states that the user interface itself can be so complicated that its testing must be automated. Second, and sadly more common, reason noted by Meszaros (2003) and Marick (1997) is that if the system has not been initially build with testability in mind GUI might be the only available interface. Whatever the reason, testability enhancements in the GUI can make automated testing easier. For example recognizing widgets can be eased assigning them a unique and unchanged identified (Kaner et al., 2001; Pettichord, 2002), and if custom controls or other widgets are not compatible with selected tool they should get special testability hooks (Kaner, 1997; Pettichord, 2002).

2.7.2 Visibility

Visibility can often be achieved through the same interface which is used for control and similarly there sometimes is a need to extend this interface to increase its testability. For example access into internals of the tested system greatly increases the visibility. This kind of access may be used for checking intermediate results of longer processes or getting some other detailed information about the state of the system.

Other techniques to increase visibility, presented by Pettichord (2002), include verbose output, logging, assertions, diagnostics and monitoring. Log files themselves can have high or low visibility. Pettichord (2002) suggests having variable levels of logging and that log messages should have timestamps and identify the subsystem which produced them. He also warns about logging too much (hard to find important messages, takes a lot of space) or too little (not enough details). Log files, as well as other outputs, should also be easily machine readable. Machine readable outputs are of course desirable also for other than testability reasons. For example easily processed log file format does not only ease automated testing but also makes it possible to automatically monitor system's state even in production environment. Another example is that having exported data in standard format like CSV or XML makes using it in testing easy but also helps further processing of the data with other tools.

2.8 Roles

This section lists and explains roles needed when building and using larger test automation frameworks. List of roles is adapted from Zallar (2001) and Fewster and Graham (1999), and it should be at least fine tuned based on the context. In smaller projects the same people may play multiple roles and in larger ones there can be several full time people per one role.

2.8.1 Test Automation Manager

Test automation projects need managers who are responsible for big decisions like starting the project and also discontinuing it if it seems to never pay back. It is possible that a steering group acts as a sponsor like this. Management is needed also in project execution level keeping the project on track. The same person can act both as a sponsor and a project manager but these tasks can well be given to separate persons. (Fewster and Graham, 1999)

2.8.2 Test Automation Architect

Test automation architect, sometimes also called a champion, is responsible for designing the framework, implementing core components and maintaining them when tested system changes or new features are needed. Architect also helps test automators implementing system specific test libraries and driver scripts. Other responsibilities include setting coding and naming conventions and writing documentation, instructions and training material for test engineers and other people involved. (Zambelich, 1998; Fewster and Graham, 1999; Zallar, 2001)

This is the most important role for project success. The person in this position has a lot of responsibilities and successfully filling them requires plenty of skills. He or she must have good programming skills, preferably from multiple programming languages, and also quite a lot of designing skills. This person must also have testing skills and be interested about it. First hand testing experience is not absolutely mandatory but highly recommended. Automation champion must, of course, have plenty of knowledge of test automation frameworks—reading the literature in the bibliography of this thesis should help but real knowledge is only gained through experience. (Fewster and Graham, 1999; Zallar, 2001)

Often there is nobody with required skills available for this role in the company. Everyone may be busy with their own projects, test engineers lack programming skills or developers do not have needed testing skills or experience from automation frameworks. Zambelich (1998) suggests that in this situation hiring a consultant with skills and experience is a good idea but, as Kaner (1997) warns, it has a risk that all deeper knowledge of the framework is in external heads. Thus it is recommended to either pair contractors with internal persons or use them only as trainers.

2.8.3 Test Automator

Test automators are mainly responsible for writing system specific driver scripts and test libraries. They must have adequate programming skills and know the framework well enough to be able to use test libraries and other reusable components it provides. They of course need to know the tested system very well because driver scripts and test libraries are always dependent on it. (Zallar, 2001; Rice, 2003)

In my experience developers who have implemented the tested system are often best candidates to this role because they have both programming skills and know the tested system by heart. Test engineers responsible for testing the system are another good option, assuming that they have enough programming skills.

2.8.4 Test Designer

End users of an automation framework are test designers. They are most often professional test engineers knowing the tested system well but in some cases they can also be domain experts with minimum testing experience. In either case they cannot be expected, or required, to have much programming skills. Framework should be designed and implemented so well that these people can easily learn how to use it and get their job done. (Kaner, 1997; Mugridge and Cunningham, 2005)

It is also possible to further divide end users' responsibilities so that more experienced test engineers both design and execute test cases while less experienced are only executing them. Framework should thus allow executing earlier created tests without needing to know how they actually are designed.

2.9 Detailed Requirements

The high level requirements for large scale test automation frameworks were defined in Table 2.1 at the beginning of this chapter. Previous sections have discussed different techniques, methods and other issues how to fulfill these requirements and at the same time split them into more detailed ones. These detailed requirements are now summarized in Table 2.3.

The list is rather long but fulfilling many of the requirements is actually pretty straightforward. For example having multiple logging levels is a feature which needs to be implemented and writing documentation is a task to undertake. The list should not be taken as a complete set of requirements covering everything but rather as an initial requirement set that needs to be adapted to a particular situation.

High Level Requirements	The framework MUST execute test cases automatically. That includes also for example verifying results, handling errors and reporting results.
	The framework MUST be easy to use without programming skills.
	The framework MUST be easily maintainable.
Automatic Test Execution	The framework MUST be able to execute tests unattended.
	It MUST be possible to start and stop test execution manually.
	It SHOULD be possible to start test execution automatically at predefined time.
	It SHOULD be possible to start test execution automatically after certain events.
	Non-fatal errors caused by the SUT or the test environment MUST be handled gracefully without stopping test execution.
	Test results MUST be verified.
	Every executed test case MUST be assigned either Pass or Fail status and failed test cases SHOULD have a short error message explaining the failure.
	Framework SHOULD differentiate expected failures (i.e. known problems) from new failures.
	Test execution MUST be logged.
	Test execution SHOULD be logged using different, configurable logging levels.
	Test report MUST be created automatically.
	Test report SHOULD be published automatically.
Ease of Use	The framework MUST use either data-driven or keyword-driven approach.
	The framework SHOULD use both data-driven and keyword-driven approach.
	Keyword-driven framework SHOULD provide means to create new higher level keywords from lower level keywords.
Maintainability	The framework MUST be modular.
	The framework SHOULD be implemented using high level scripting languages.
	The testware MUST be under version control.
	The framework MUST have coding and naming conventions.
	The framework MUST be adequately documented.
	Testability of the tested system MUST be increased as much as possible.
	Clear roles MUST be assigned.

Table 2.3: Detailed requirements for a large scale test automation framework

2.10 Chapter Summary

This chapter first defined that high level requirements for large scale test automation frameworks are automatic test execution, ease of use and maintainability. Then various issues related to how to fulfill the requirements were discussed, ranging from needed features to implementation issues. Finally more detailed requirements were summarized in Table 2.3.

Next chapter builds from the foundation laid in this chapter and suggests a high-level concept for a framework fulfilling the defined requirements.

Chapter 3

Concept for Large Scale Test Automation Frameworks

The previous chapter defined requirements for large scale test automation frameworks and this chapter continues by presenting a framework concept that tries to fulfill those requirements. The presented concept should, however, not to be considered universally suitable for all situations as is. Instead it is more like a high level layout that needs to be adjusted based on the context where it is used.

3.1 Framework Structure

The requirement for the framework to be either data-driven, keyword-driven, or both means that some kind of system where test data can be designed and created is needed. Multiple requirements for monitoring test execution suggests that another system is needed for monitoring purposes and yet another system is, of course, needed for actually executing test cases. In high level these three must-have systems make up the test automation framework as seen in Figure 3.1. Test design and monitoring systems can of course also be integrated to make all user interfaces available in one place.

3.1.1 Test Design System

Test design system is used for creating new test cases and editing existing ones. It is test engineers' and domain experts' playground and must be easy to use with minimal training and without programming skills.

As discussed in Section 2.4.2, the created test data can be stored into flat files or databases. A simple solution is using an existing tool like spreadsheet program or HTML editor as a test design system. A special test design tool made just for this purpose would of course have its benefits but building something like that is out of the reach for most automation projects.

3.1.2 Test Monitoring System

Test monitoring system is used for controlling test execution and checking test results. To fulfill requirements in Table 2.3 it should have at least the capabilities listed below.

- Starting test execution manually.
- Starting test execution automatically after some specified event (e.g. new version of the SUT available for testing).
- Starting test execution automatically at specified time.
- Stopping test execution.
- Monitoring test execution while tests are running.
- Viewing test logs while tests are running and afterwards.
- Viewing test report.
- Changing logging level.

The low-end test monitoring solution is using command line for starting and stopping test execution and creating plain text test logs and reports. Next step is creating logs and reports in HTML which is much richer than plain text but equally universally viewable. The high-end solution could be having all test monitoring functionalities in a web based system and integrating it also with the test design system.

3.1.3 Test Execution System

Test execution system is the core of the framework. Its main components are driver scripts, the test library, the test data parser and other utilities like logger and report generator. How all these and other framework components work together is illustrated in Figure 3.2. Arrows in the diagram start from the component that uses the other component. They do not mean that the interaction is only one way, however, as the used component always returns something to the caller. For example the parser returns processed test data and test functions return the test status and the status message.

Driver Scripts

Test execution is controlled by driver scripts which are started using the test monitoring system. As discussed in Section 2.3.2, driver scripts are pretty short and simple because they mainly use services provided by test libraries and other reusable modules. This can also be seen in Figure 3.2 which has plenty of arrows starting from the driver script denoting that it is using the component at the other end.

Test Libraries

Test libraries contain all those reusable functions and modules which are used in testing or in activities supporting it. Testing is mainly interacting with the tested system and checking that it behaves correctly. Supporting activities are often related to setting up the test environment and include tasks like copying files, initializing databases and installing software. Functions in the test library can do their tasks independently but they can also use other functions or even external tools—all these three options are seen in Figure 3.2.

As explained in Section 2.3.2 one test library can be used by several driver scripts. If the system under test is very large, or framework is designed for testing multiple different systems, it could be a good idea to create one test library for each tested system or subsystems.

Test Data Parser

The test data parser was introduced in Section 2.4.3 as a means to get test data easily to driver scripts. Its task is processing the test data and returning it to the driver script in easy to use test data containers. Test data containers provide an easy and presentation neutral access to the test data.

The role of the parser in Figure 3.2 may seem small but in reality it is the heart of the whole test execution system. Extending the metaphor test data containers are blood cells carrying oxygen to muscles—only this time oxygen is test data and muscles are functions in the test library. Finally the driver script controlling everything can be seen as a brain of the framework.

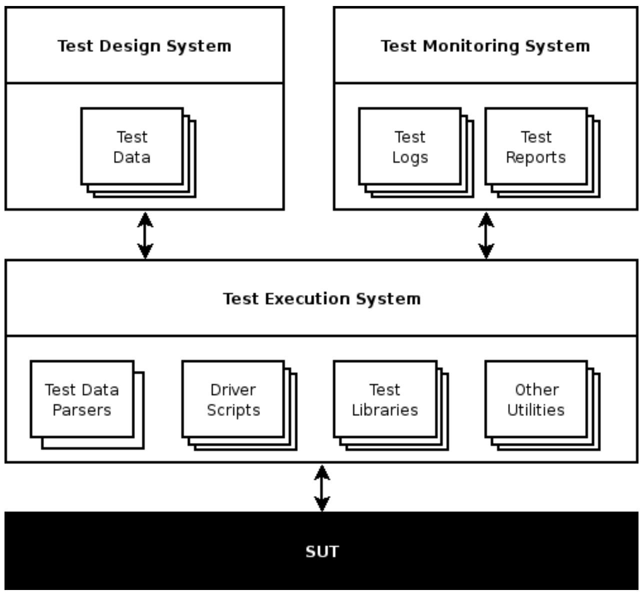


Figure 3.1: High level view of the framework

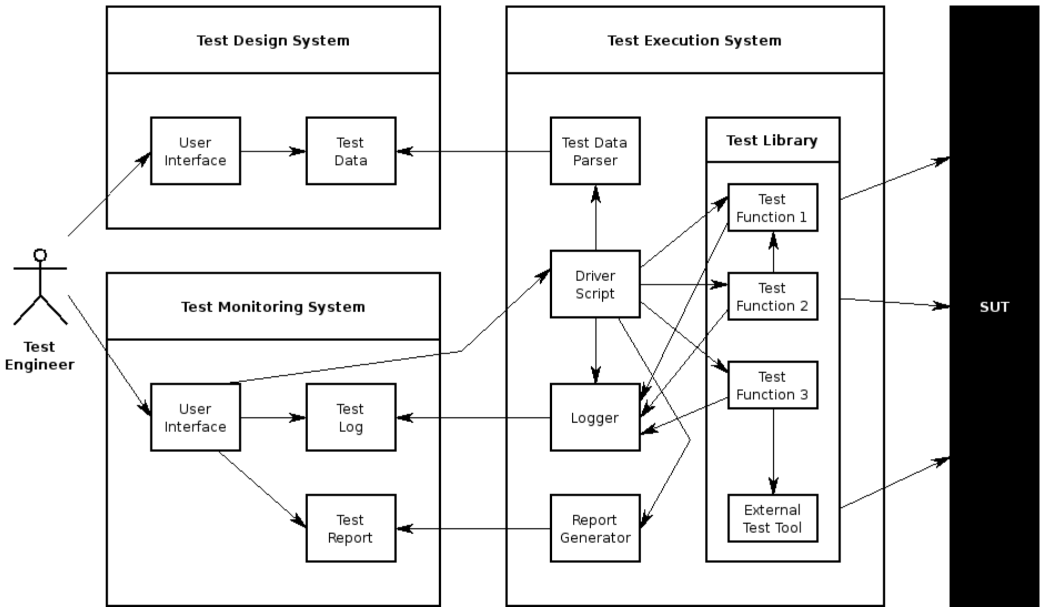


Figure 3.2: Detailed view of the framework

Processing data-driven and keyword-driven test data requires two different parser flavors. Data-driven parser and test data containers created by it are described in Section 3.2 and keyword-driven versions in Section 3.3.

Logger

Section 2.2.7 introduced quite a few logging requirements to the framework. An easy and modular way to fulfill them is having a common logger component with the required features. This is illustrated in Figure 3.2 where both the driver script and test functions use the same logger which in turn writes messages to the common test log.

Report Generator

The reporting requirements of the framework are defined in Section 2.2.8 and they can be summarized saying that a concise report must be created and published. These tasks are given to a reusable report generator component.

3.2 Presenting and Processing Data-Driven Test Data

At this point the framework has a set of requirements and a layout which helps to achieve them. Section 3.1.3 highlighted the importance of the test data parser and test data containers it creates and dubbed them framework's heart and blood cells. This section explains how the data-driven version of the parser works and the keyword-driven version is discussed in the next section.

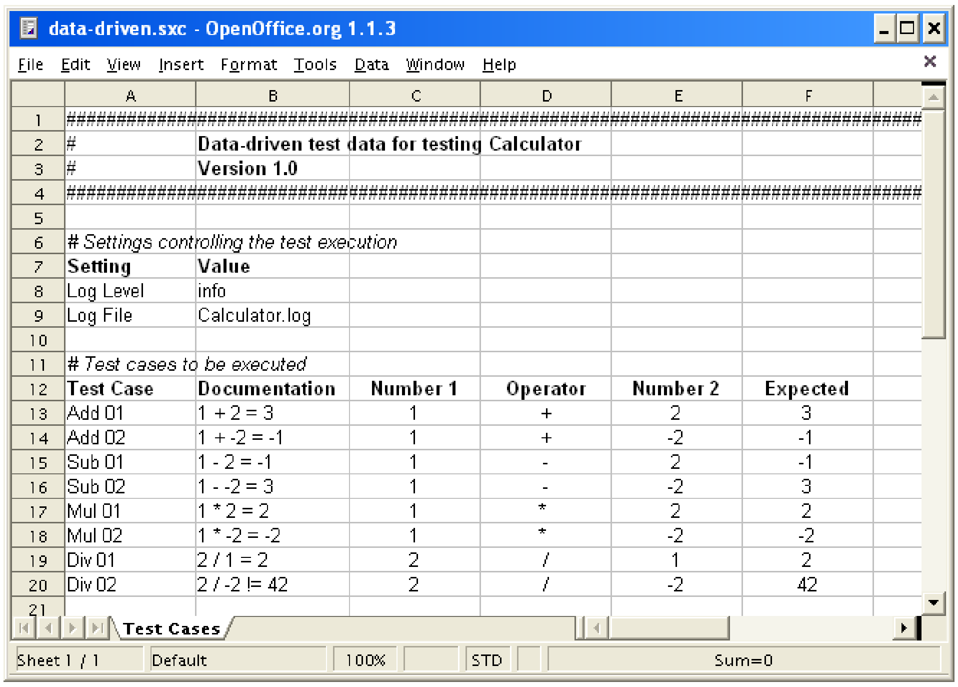
The idea of having a separate parser and letting it create test data containers is not directly from literature. It is based on my experiences from designing and implementing a larger data-driven framework. Many of the original ideas were of course borrowed from books and articles I had read earlier, and the now presented version has been refined after I have studied the subject further. The parser component conforms very well to similar components in frameworks presented by Nagle (2000), Buwalda et al. (2002), Zambelich (1998) and Mosley and Posey (2002). Test data containers, on the other hand, are something I have not seen presented elsewhere. The framework in general is also pretty similar to Fit (Framework for Integrated Testing) presented by Mugridge and Cunningham (2005) but that is mainly a coincidence as I studied it more only after the framework was designed.

3.2.1 Presenting Test Cases

The layout and formatting of the test data creates a simple test description language. Test designers must know how the languange is "programmed" and the associated driver script must know how to parse and execute it. This language must be designed cooperatively with test designers and test automators—the former knows what is needed and the latter what is feasible technically. Figure 3.3 shows one way of presenting data-driven test data and different parts of it are explained below. Depending on the system under test, selected test design system and other issues there are of course different suitable alternatives.

Test Case Name

The name of the test case is specified in the column with title Test Case. When the data is parsed the name is given to the created test data container. The driver script and other components can later query the name from the container and use it when logging and reporting.


data-driven.sxc - OpenOffice.org 1.1.3
File Edit View Insert Format Tools Data Window Help
Test Cases
Sheet 1 / 1
Default
100%
STD
Sum=0

	A	B	C	D	E	F
1	##					
2	#	Data-driven test data for testing Calculator				
3	#	Version 1.0				
4	##					
5						
6	# Settings controlling the test execution					
7	**Setting**	**Value**				
8	Log Level	info				
9	Log File	Calculator.log				
10						
11	# Test cases to be executed					
12	**Test Case**	**Documentation**	**Number 1**	**Operator**	**Number 2**	**Expected**
13	Add 01	1 + 2 = 3	1	+	2	3
14	Add 02	1 + -2 = -1	1	+	-2	-1
15	Sub 01	1 - 2 = -1	1	-	2	-1
16	Sub 02	1 - -2 = 3	1	-	-2	3
17	Mul 01	1 * 2 = 2	1	*	2	2
18	Mul 02	1 * -2 = -2	1	*	-2	-2
19	Div 01	2 / 1 = 2	2	/	1	2
20	Div 02	2 / -2 != 42	2	/	-2	42
21						

Figure 3.3: Example data-driven input file

Documentation

Test case's documentation is specified in the next column. It obviously documents the test data but that is not its only purpose. Since it is stored to the test data container it is available also at test execution time and can be written to test logs and reports. Python programmers may find the idea similar to how modules, classes and functions are documented in Python.

Test Data

Actual test data is put into subsequent columns with appropriate column names. The data is again stored to the container and can be queried from it when the name of the column is known.

Settings

The most important part of the test data is of course a list of test cases and test data associated with them. I have noticed, however, that it is often useful to have a possibility to specify also settings controlling the test execution along with the data. Settings can be generic (e.g. logging level and log file) or specific to the system under test (e.g. IP address of a machine to use in tests).

Comments, Formatting and Empty Lines

Figure 3.3 shows how comment lines, marked with a hash character (#), can be used to document the purpose of the whole file or to comment specific parts of it. In general comments make the test data easier to understand. Similar effects are achieved also with empty lines separating blocks of data and formatting (e.g. bold and italic) helping to visualize the data.

All these features are unquestionably important as they increase both usability and maintainability of the test data. In test execution phase they are not anymore needed, however, and the parser should ignore them.

Handling Parsing Errors

If the test data is invalid or any errors occur during parsing, the parser must not allow test execution with corrupted test data because that is likely to cause hard

```
import DataDrivenParser

testData = DataDrivenParser.parse('data-driven.sxc')
logLevel = testData.getSetting('Log Level')
testCases = testData.getTestCases()

for testCase in testCases:
    name = testCase.getName()
    doc = testCase.getDoc()
    number1 = testCase.getData('Number 1')
    operator = testCase.getData('Operator')
    number2 = testCase.getData('Number 2')
    expected = testCase.getData('Expected')
    # Use the test data for testing
```

Listing 3.1: Using data-driven test data

to diagnose errors later on. Parser may either terminate immediately after the first error, or continue parsing to find and report all possible errors. In either case it must provide a clear error message telling what is wrong and where.

3.2.2 Using Test Data

The test data parser does not return the test data to driver scripts in rows and columns as it is presented. Instead the parser creates generic test data containers and stores processed test data into them. These containers are then returned to the driver script which can use methods provided by them to query the test data when it is needed. Listing 3.1 shows how driver scripts can get test data from the parser and use returned containers.

Test Data Container

A test data container contains all the presented test data i.e. both test cases and settings. In Listing 3.1 a test data container is returned by the imported parser and assigned to the `testData` variable. It can then be queried to get settings and, more interestingly, a list of test case containers.

It is a good idea to implement the `getSetting` method so that it accepts a default argument which is returned if the queried setting is not defined. That way tests can be run even if the value is not specified along the test data but the default can be overridden easily. In Listing 3.1 that would change getting log level for example into `testData.getSetting('Log Level', 'info')`.

Test Case Container

A test case container has all the test data of one particular test case. As it was seen in Section 3.2.1 that data includes test case's name, documentation and the actual test data. Name and documentation are always present so they can be queried with special methods (`getName` and `getDoc`) but other data must be queried using a generic `getData` method with the name of the needed data as an argument.

Summary of Data Containers and Their Methods

Table 3.1 summarizes presented test data containers and their methods. If the framework needs more capabilities it is possible to add additional methods to the containers and make them more intelligent. For example method like `getIntData` could be used to get the requested data as an integer.

Test Data Container	`getTestCases()`
	`getSetting(name, default)`
	(default is optional)
Test Case Container	`getName()`
	`getDoc()`
	`getData(name)`

Table 3.1: Test data containers and methods needed in data-driven testing

3.2.3 Example

It is a time for an example putting together features needed when using data-driven test data. This example executes tests based on the test data in Figure 3.3. Listing 3.2 shows an example driver script which uses a test library in Listing 3.3. Created test log is shown later in Figure 3.4.

The system under test in this example is a simple calculator. In real life it would probably be an external system used through special tool but in this case the calculator is just imported and used directly in the test library. More realistic examples are presented in the next chapter.

```
# Import reusable components
import DataDrivenParser
import TestLibrary
import TestLog

# Parse test data
testData = DataDrivenParser.parse('testdata.tsv')

# Get log level, initialize logging and get a logger
logLevel = testData.getSetting('Log Level')
TestLog.initialize('testlog.log', logLevel)
logger = TestLog.getLogger('Driver')

# Log a message with info level
logger.info('Test execution started')

# Variables collecting test statistics
executed = 0
passed = 0

# This loop is iterated for each test case
for tc in testData.getTestCases():
    executed += 1
    name = tc.getName()
    doc = tc.getDoc()
    logger.info('Started test case %s (%s)' % (name, doc))

    # Try running the test. If no errors occur test's status and
    # a status message are returned.
    try:
        (status, message) = TestLibrary.testFunction(tc)

    # Catch possible exceptions.
    except Exception, error:
        status = False
        message = str(error)

    # Log test status either as Pass or Fail.
    if status is True:
        logger.passed('Test case %s passed' % name)
        passed += 1
    else:
        logger.failed('Test case %s failed: %s' % (name, message))

logger.info('%d test cases executed, %d passed, %d failed' % \
            (executed, passed, executed - passed)
```

Listing 3.2: Data-driven driver script example

```
import TestLog
import Calculator  # The system under test

# Get own logger for the library
logger = TestLog.getLogger('Library')

def testFunction(tc):

    # Get test data. Note that numeric data is turned into integers
    # (getIntData method would be convenient)
    number1 = int(tc.getData('Number 1'))
    operator = tc.getData('Operator')
    number2 = int(tc.getData('Number 2'))
    expected = int(tc.getData('Expected'))

    # Debug logging helps investigating possible problems
    logger.debug('Num 1: %d, Operator: %s, Num 2: %d, Expected: %d' \
                 % (number1, operator, number2, expected))

    # Test the calculator
    Calculator.input(number1)
    if operator == '+':
        Calculator.add(number2)
    if operator == '-':
        Calculator.subtract(number2)
    if operator == '*':
        Calculator.multiply(number2)
    if operator == '/':
        Calculator.divide(number2)

    result = Calculator.getResult()

    # Check test result
    if result == expected:
        status = True
        message = 'Test passed. Result: %d' % (result)
        log.info(message)
    else:
        status = False
        message = 'Expected: %d, but got: %d' % (expected, result)
        log.failure(message)

    return status, message
```

Listing 3.3: Data-driven test library example

Timestamp	**Module**	**Level**	**Message**
20050509 12:23:34	Driver	Info	Test execution started.
20050509 12:23:34	Driver	Info	Started test case Add 01 (1 + 2 = 3)
20050509 12:23:35	Library	Info	Test passed. Result: 3
20050509 12:23:35	Driver	Pass	Test case Add 01 passed.
20050509 12:23:35	Driver	Info	Started test case Add 02 (1 + -2 = -1)
⋮	⋮	⋮	⋮
20050509 12:23:40	Driver	Pass	Test case Div 01 passed.
20050509 12:23:41	Driver	Info	Started test case Div 02 (2 / -2 != 42)
20050509 12:23:41	Library	Failure	Expected: 42, but got: -1
20050509 12:23:41	Driver	Fail	Test caseDiv 02 failed: Expected: 42, but got: -1
20050509 12:23:42	Driver	Info	8 test cases executed, 7 passed, 1 failed

Figure 3.4: Data-driven test log example

3.3 Presenting and Processing Keyword-Driven Test Data

The close relationship between keyword-driven and data-driven techniques makes presenting and processing test data very similar in both cases. Many of the ideas introduced in the previous section are thus valid also with keyword-driven test data. There is a totally new challenge, though, as the framework should support constructing user keywords.

Because keyword-driven approach is more flexible than data-driven also test description language it provides is more expressive. For example creating new user keywords with arguments is getting quite close to real programming. This kind of features make the framework more powerful but also more complex. Just adding more functionality like this is necessarily not a good thing as one of the main ideas behind data-driven approaches is making test design simple and not requiring programming skills. The presented syntax is still so simple, however, that learning it should not be a problem for anyone involved with test automation. I believe a right design goal in this kind of situations is keeping easy things easy but making complicated things possible.

3.3.1 Presenting Test Cases

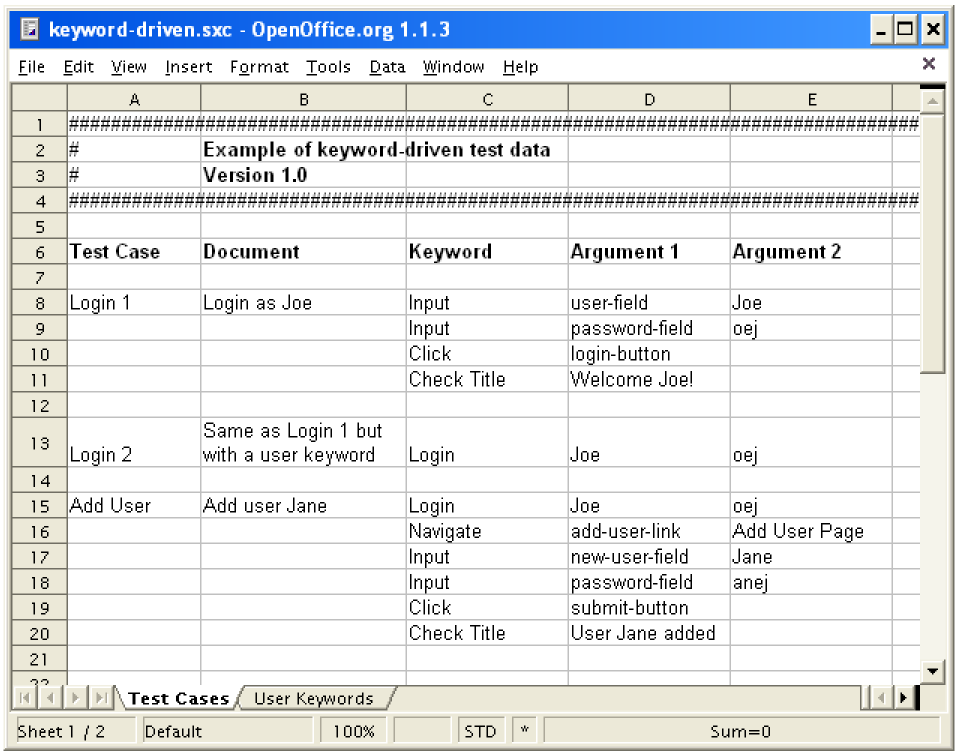
Figure 3.5 shows one possibility how keyword-driven test data can be presented. The major difference between this example and earlier data-driven example in Figure 3.3 is that instead of having one test case per row there is now one keyword per row. The number of keywords per test case may vary and nothing prevents test cases from having totally different keywords.

Name and Documentation

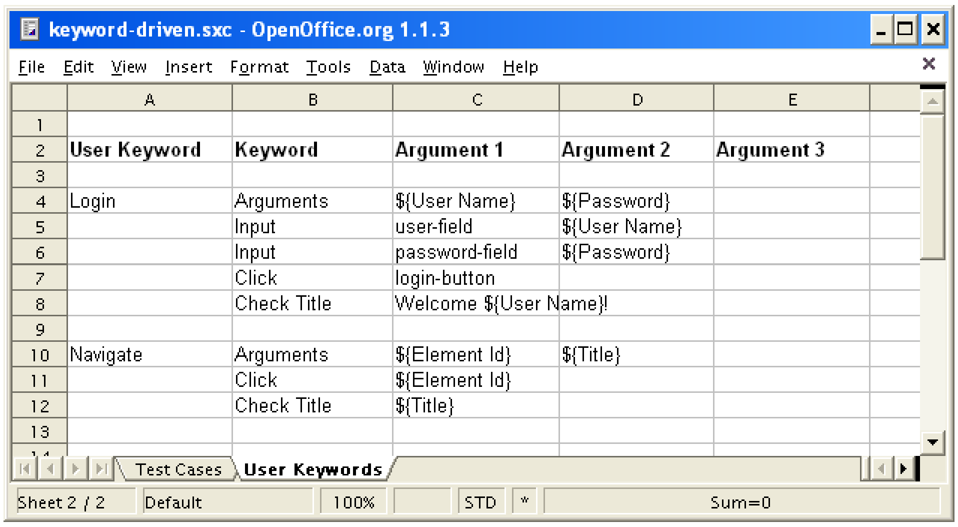
Test case's name and documentation are used exactly the same way as they are used with data-driven test data.

Keyword and Arguments

Keywords and their arguments are something completely new compared to data-driven test data. Keyword refers either to a base or user keyword which both can be used freely when test cases are constructed. Keywords have variable number of arguments and rest of the columns are reserved for them.


keyword-driven.sxc - OpenOffice.org 1.1.3
File Edit View Insert Format Tools Data Window Help
Test Cases
User Keywords
Sheet 1 / 2
Default
100%
STD
Sum=0

	A	B	C	D	E
1	##				
2	#	**Example of keyword-driven test data**			
3	#	**Version 1.0**			
4	##				
5					
6	**Test Case**	**Document**	**Keyword**	**Argument 1**	**Argument 2**
7					
8	Login 1	Login as Joe	Input	user-field	Joe
9			Input	password-field	oej
10			Click	login-button	
11			Check Title	Welcome Joe!	
12					
13	Login 2	Same as Login 1 but with a user keyword	Login	Joe	oej
14					
15	Add User	Add user Jane	Login	Joe	oej
16			Navigate	add-user-link	Add User Page
17			Input	new-user-field	Jane
18			Input	password-field	anej
19			Click	submit-button	
20			Check Title	User Jane added	
21					


keyword-driven.sxc - OpenOffice.org 1.1.3
File Edit View Insert Format Tools Data Window Help
Test Cases
User Keywords
Sheet 2 / 2
Default
100%
STD
Sum=0

	A	B	C	D	E
1					
2	**User Keyword**	**Keyword**	**Argument 1**	**Argument 2**	**Argument 3**
3					
4	Login	Arguments	${User Name}	${Password}	
5		Input	user-field	${User Name}	
6		Input	password-field	${Password}	
7		Click	login-button		
8		Check Title	Welcome ${User Name}!		
9					
10	Navigate	Arguments	${Element Id}	${Title}	
11		Click	${Element Id}		
12		Check Title	${Title}		
13					

Figure 3.5: Keyword-driven test cases and user keywords

Settings

Even though Figure 3.5 does not have any settings they could be used similarly as they are used with data-driven test data.

Comments, Formatting and Empty Lines

Comments, formatting and empty lines make both data-driven and keyword-driven test data easier to use and maintain. In both cases they should also be ignored by the parser.

Handling Parsing Errors

Both data-driven and keyword-driven parsers must terminate if the test data they process is corrupted. Because keyword-driven test data is somewhat more complex there can also be more errors and the parser must be better at catching and reporting them. It should for example check that keywords used in tests actually exist.

3.3.2 Presenting User Keywords

Constructing new "virtual" user keywords from base keywords implemented in the test library (and already created user keywords) is one of the most important features of the presented framework concept. Because higher level keywords can be created freely, keywords in the test library can be responsible only for atomic tasks. That, in turn, makes implementing and maintaining base keywords easy.

Creating user keywords is an integral part of the keyword-driven test description language. Figure 3.5 shows one approach which should not be too hard to understand. This is certainly an area where a real test design tool could have a superior user interface compared to a spreadsheet program.

User Keyword Name

User keyword's name specifies how to call it in test cases or other user keywords. User keywords could also have similar documentation as test cases have.

Keyword and Arguments

User keywords are constructed from other keywords which can be both base keywords and other user keywords. Keywords are listed one per row similarly as when designing test cases.

User keywords must be able to use arguments similarly as base keywords. In Figure 3.5 arguments are specified using a special Arguments keyword. Arguments themselves look like ${ArgumentName} to make them stand out from literal text. They can be used freely inside the user keyword. When the keyword-driven parser processes the test data it substitutes the string ${ArgumentName} with the value specified when the keyword is used. For example when Login keyword is used in a test case with arguments "Joe" and "eoj" the following things happen.

1. User keyword **Login** is called with arguments *Joe* and *eoj*
2. Base keyword **Input** is called with arguments *user-field* and *Joe*
3. Base keyword **Input** is called with arguments *password-field* and *eoj*
4. Base keyword **Click** is called with argument *login-button*
5. Base keyword **Check Title** is called with argument *Welcome Joe!*

This is illustrated in Figure 3.5 where test cases Login 1 and Login 2 end up doing exactly same things. That pretty well demonstrates how user keywords make tests shorter and easier to read. Another nice benefit is that the same base keywords which were used for creating the Login keyword can be used to create multiple other user keywords. In general only a small set of low level base keywords is needed for testing even a complex system.

3.3.3 Using Test Data

The keyword-driven parser creates similar containers from the test data as its data-driven counterpart. The difference is that now the test data container does not contain test data directly but a list of keyword containers. How keyword-driven test data can be used is shown in Listing 3.4 and required containers are explained afterwards.

```
import KeywordDrivenParser
import TestLog
import TestLibrary

testData = KeywordDrivenParser.parse('testdata.tsv')
logLevel = testData.getSetting('Log Level', 'info')
logFile = testData.getSetting('Log File', 'testlog.log')
TestLog.initialize(logFile, logLevel)
logger = TestLog.getLogger()


def executeKeyword(keyword):
    name = keyword.getName()
    if keyword.getType() == 'user'
        logger.debug('Executing user keyword %s' % name)
        for subKeyword in keyword.getKeywords():
            executeKeyword(keyword)
    else:
        logger.debug('Executing base keyword %s' % name)
        handlerName = name.lower().replace(' ', '') + 'Handler'
        if not hasattr(TestLibrary, handlerName):
            raise Exception, 'No handler for keyword %s' % name
        handler = getattr(TestLibrary, handlerName)
        args = keyword.getArgs()
        handler(*args)


## Main loop
for testCase in testData.getTestCases():
    logger.info('Executing test case %s (%s)' % (testCase.getName(),
                                                  testCase.getDoc()))
    keywords = testCase.getKeywords()
    logger.debug('Test case has %d keywords' % len(keywords))

    for keyword in keywords:
        try:
            executeKeyword(keyword)
        except:
            ...
        # Catch possible exceptions, assign test statuses, etc.
```

Listing 3.4: Using keyword-driven test data

```
import TestLog

class TestLibrary:

    def __init__(self, name):
        self.logger = TestLog.getLogger(name)

    def logHandler(self, message, level='info'):
        self.logger.write(message, level)

    def inputHandler(self, fieldId, text):
        # Somehow input the given text to field with given id

    def checkresultHandler(self, expected):
        # Check that result matches expected results
```

Listing 3.5: Very simple test library

Test Data Container

The same test data container which is used by the data-driven parser can be used also by the keyword-driven parser. In both cases the test data container contains test cases and settings and has methods for querying them.

Test Case Container

Test case containers used by different parsers differ but they both have methods for getting the name and documentation. The keyword-driven version does not have any test data itself. Instead it has a list of keyword containers and a `getKeywords` method for getting them.

Keyword Container

There are two types of keywords so there must also be two different keyword containers. Both of these have a name and a type which can be queried with `getName` and `getType` methods. User keyword containers also have a list of keywords and `getKeywords` method similarly as test case containers. Base keyword containers carry the actual test data, keyword's arguments, and provide access to them with `getArgs` method.

The method `executeKeyword` in Listing 3.4 shows a simple pattern how keywords can be executed. The first part is figuring out whether the given keyword is of base or user keyword type using the `getType` method. User keywords are handled simply by recursively calling `executeKeyword` again for all its sub keywords. Handling base keywords is also surprisingly easy. First the name of the hander method is needed and that is got by lowercasing keyword's name, removing all spaces from it and adding text "Handler" at the end. That way for example keywords Input and Check Result get handlers `inputHandler` and `checkresultHandler`, respectively. When the name of the handler is known standard Python functions `hasattr` and `getattr` are used to first check that handler exists and then to get a reference to it (most modern programming languages have similar dynamic features). Last part of the execution is getting arguments specified in the data and calling the handler method with them.

Summary of Container Objects and Methods

Table 3.2 summarizes explained test data containers and their methods. Similarly as with data-driven testing it is possible to add more features to containers if needed.

Test Data Container	`getTestCases()`
	`getSetting(name, default)`
	(default is optional)

Test Case Container	`getName()`
	`getDoc()`
	`getKeywords()`

User Keyword Container	`getName()`
	`getType()`
	`getKeywords()`

Base Keyword Container	`getName()`
	`getType()`
	`getArgs()`

Table 3.2: Test data containers and methods needed in keyword-driven testing

3.4 Chapter Summary

This chapter has introduced a high-level concept for data-driven and keyword-driven frameworks that consists of test design, monitoring and execution systems. The first two systems are used by for designing test cases and controlling and monitoring their execution. The test execution system, the core of the framework, is used only indirectly through the test monitoring system.

After explaining the framework structure this chapter discussed how the data-driven and keyword-driven test data can be presented to users and processed by the framework. Both presenting and processing the data-driven test data is pretty straightforward but achieving the greater flexibility of keyword-driven requires more from the framework.

In the next chapter a pilot is conducted to verify whether the presented framework concept is usable in real testing.

Chapter 4

Implementation and Pilot

Previous chapter suggested a concept for large scale test automation frameworks. The concept includes a layout describing what components are needed and an explanation of how data-driven and keyword-driven test data can be presented to users and processed by the framework. This chapter continues from that and describes how prototypes of the needed key components were implemented and how they were used to test two different software systems. The framework concept is then evaluated based on the pilot experiences in the next chapter.

4.1 Implementation Decisions

4.1.1 Technical Decisions

Storing Test Data

First decision to make was how to edit and store the test data. Building a special test design tool would have been a really big task so, as Section 2.4.2 suggests, I decided to use a spreadsheet program. Because I wanted to avoid problems with TSV and CSV file formats I needed to parse the programs native file format. I had two options to choose from: Microsoft Excel and OpenOffice Spreadsheet. OpenOffice's greater platform coverage and open, standardized file format were such huge pluses that the final decision was pretty easy.

Implementation Language

Next decision was to select the implementation language. I know Java, Perl and Python fairly well so it needed to be one of them. As discussed in Section 2.6.1, scripting languages are most often good for this kind of work so I dropped Java. Based on my earlier experience I chose Python mainly because I feel that code written with it is generally clearer and easier to maintain than Perl code. I also find Python's object oriented features better.

After selecting Python I needed to make one more choice: should I write the code so that it is Jython compatible or not. Jython is a Java implementation of Python and it integrates seamlessly with Java. Since it can interact directly with Java APIs it is an excellent tool for testing Java applications and also for testing using Java tools like HTTPUnit. Only problem with Jython is that it is a bit behind the standard Python in versions and latest Python features do not work in Jython. I felt having Jython compatibility so important that I decided to stick with Jython compatible Python features.

4.1.2 Decisions Regarding the Pilot

Systems Under Test

Unfortunately during the thesis writing process I did not have a possibility to test the framework concept in a real project. To make up that shortcoming and to get a broader view about framework's capabilities and shortcoming I decided to use it for testing two different systems.

The first tested system I selected was a Windows application, represented by the standard Windows Calculator seen in Figure 4.1. If testing Calculator succeeds, testing other Windows applications ought to be possible also.

The second system under test was web sites in general. I decided to make a Google search—everyone's favorite web testing example—to test filling and submitting forms and use web pages of Helsinki University of Technology to verify that basic navigation works. I was sure that if these tests are easy to implement testing more complex web sites and applications would also be possible.

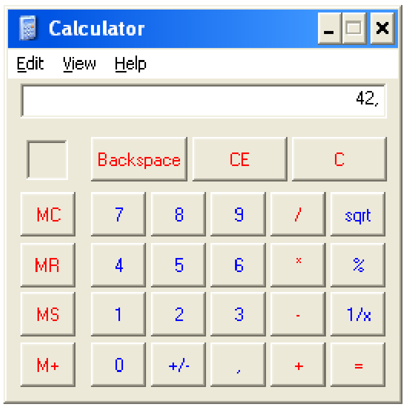


Figure 4.1: Standard Windows Calculator

Testing Approaches

Since the framework can be used both for data-driven and keyword-driven testing I wanted to test both of these testing methods. However, I did not believe testing both of the selected systems using both approached was worth the effort and decided to drop one of the four possibilities. The one I excluded was data-driven web testing. The main reason was that it did not suit well for that kinds of tests I had planned, but I had also started to feel that keyword-driven testing is the more interesting approach and wanted to use it with both systems.

In the end I had selected following three combinations for the pilot.

- Data-driven Windows application testing
- Keyword-driven Windows application testing
- Keyword-driven web testing

4.2 Implementing Reusable Framework Components

This section explains in detail what reusable framework components were implemented and why. How the components were implemented is also discussed briefly.

4.2.1 Test Data Parser

The framework concept relies heavily on reusable components. The most important component is the test data parser which consists of multiple subcomponents. First of all it comes in two flavors but it also includes different test data containers.

I built both data-driven and keyword-driven parsers and implemented all the basic features discussed in Section 3.2 and Section 3.3, respectively. Adding more capabilities to parsers and test data containers would be easy.

OpenOffice Parser and Raw Data Containers

Parsing the test data was implemented so that it happens in two phases. In the first phase the data is simply read from the media where it is stored and in the second it is processed further by the data-driven or keyword-driven parser. This means that there must be a separate parser to take care of the first phase and also an intermediate containers for the raw test data. Handling parsing in two phases makes it easy to change the format where the test data is stored as the only new component needed is a parser for that media.

I had already decided to use OpenOffice spreadsheet so I needed to write a parser for it. I also needed raw data containers where the OpenOffice parser can store the data. I implemented many of the parsing features, e.g. removing comments, in raw data containers because they are more reusable than the parser for OpenOffice format. I also tried to make it as convenient as possible for the data-driven and keyword-driven parsers to get the raw data. In the end raw data containers grew in importance and also in size.

Data-Driven Parser

Data-driven parser gets the raw test data from OpenOffice parser. It then processes the data and stores the outcome to relevant test data containers. Finally test data containers are returned to the calling driver script. Because the data-driven parser gets the data in easy to use raw data containers the processing task is pretty simple.

Keyword-Driven Parser

Keyword-driven parser does exactly the same as its data-driven counterpart but for keyword-driven test data. Keyword-driven test data of course has more to process because it contains both test cases and created user keywords.

Constructing user keywords was the hardest part to implement and getting them working required few attempts and to get the code somewhat organized required few more refactorings. In the end much of the logic regarding user keywords actually went to relevant test data containers and left the parser itself pretty simple.

Test Data Containers

Test data containers include test data, test case and keyword containers. Different containers and their features are explained in the previous chapter. While implementing the keyword-driven parser these containers got a bit more functionality as they became partly responsible for creating user keywords.

Common Utilities

Some common utilities were needed by different parsers. A small amount of extra code was also needed to make parsers Jython compatible.

4.2.2 Logger

Besides test data parsers only a logger component was needed for evaluating the framework concept. For example a report generator is not really needed because it is always possible to create test reports based on the test logs and that does not have anything to do with the overall concept.

Table 4.1 lists implemented logging functionalities and shows that they cover most of the requirements defined in Section 2.2.7. Logging must first be initialized in the driver script which can also change the logging level later. Components that want to log something can get a logger and use its logging methods which map to logging levels presented in Table 2.2. Note that the methods for levels pass and fail are called `passed` and `failed`, respectively, because `pass` is a reserved work in Python and should not be used.

Test Log Module	`initialize(logFile, logLevel)`	
	`setLevel(logLevel)`	
	`getLogger(name)`	
Logger Component	`failed(msg)`	`warning(msg)`
	`passed(msg)`	`info(msg)`
	`fatal(msg)`	`debug(msg)`
	`error(msg)`	`trace(msg)`
	`failure(msg)`	

Table 4.1: Implemented logging functionality

4.2.3 Summary

Implementing needed reusable framework components was not too big a task. With all the rewrites and refactorings it took about two weeks. In calendar time it of course took a bit more as it was implemented at the same time as I designed the framework itself. Current implementation is also just a pilot and does not have all the features needed in real large scale testing. It should, however, be good enough to test whether the framework concept is functional or not.

Table 4.2 lists sizes of implemented components in lines of code so that empty and comment lines are excluded. The most interesting finding from the table is that the whole parser module is less than 500 lines in size even though it contains both the data-driven and keyword-driven parser and everything they need. In general all the reusable components are very small and total lines of code is clearly below 600. That is a very small amount of code, especially when compared to sizes of the systems which can potentially be tested with the framework.

Test Data Parser	479
OpenOffice Parser	88
Raw Data Containers	102
Data-Driven Parser	22
Keyword-Driven Parser	64
Test Data Containers	136
Common Utilities	67
Logger	82
TOTAL	562

Table 4.2: Sizes of reusable framework components

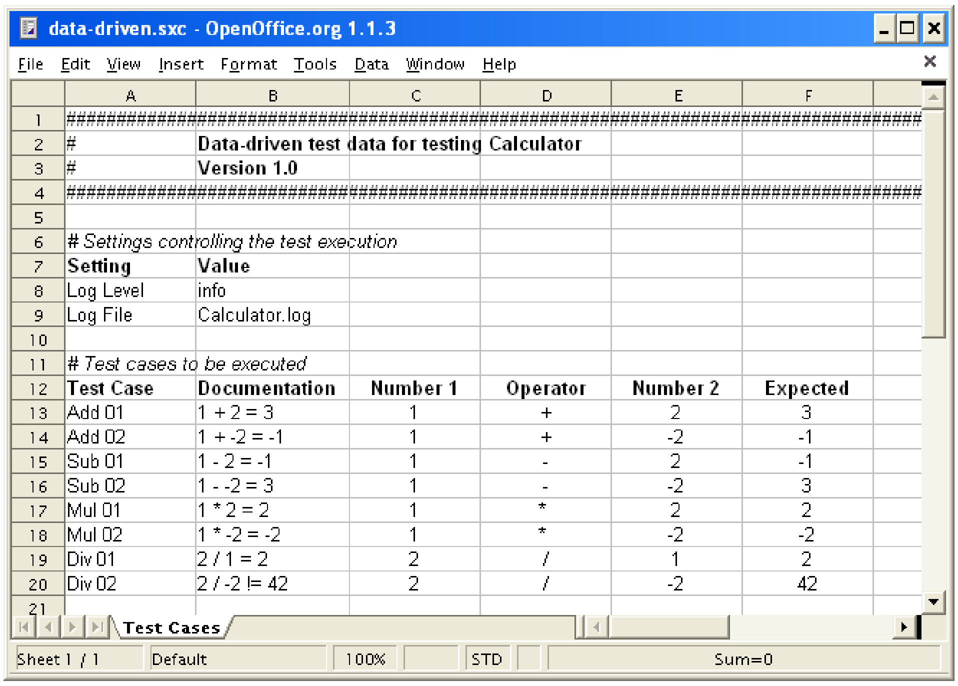
4.3 Data-Driven Windows Application Testing

4.3.1 Test Data

The test data used in Calculator pilot is shown in Figure 4.2 and it is the exactly same data as in Figure 3.3 earlier. In real testing this small number of test cases would of course not be enough but it ought to be enough for a pilot.

4.3.2 Driver Script

The driver script controlling the test executing is shown in Listing 4.1. It is somewhat different from the earlier data-driven example shown in Listing 3.2. There the test library had a test function which took care of the whole test but now the driver uses smaller functions to interact with the tested system. Another difference is that in earlier example the test library returned test status explicitly but now the status is communicated with exceptions. No exceptions means that test passed, `AssertionError` tells that some verification has failed and all other exceptions denote unexpected errors in test execution.


data-driven.sxc - OpenOffice.org 1.1.3
File Edit View Insert Format Tools Data Window Help
Test Cases
Sheet 1 / 1
Default
100%
STD
Sum=0

	A	B	C	D	E	F
1	##					
2	#	Data-driven test data for testing Calculator				
3	#	Version 1.0				
4	##					
5						
6	*# Settings controlling the test execution*					
7	**Setting**	**Value**				
8	Log Level	info				
9	Log File	Calculator.log				
10						
11	*# Test cases to be executed*					
12	**Test Case**	**Documentation**	**Number 1**	**Operator**	**Number 2**	**Expected**
13	Add 01	1 + 2 = 3	1	+	2	3
14	Add 02	1 + -2 = -1	1	+	-2	-1
15	Sub 01	1 - 2 = -1	1	-	2	-1
16	Sub 02	1 - -2 = 3	1	-	-2	3
17	Mul 01	1 * 2 = 2	1	*	2	2
18	Mul 02	1 * -2 = -2	1	*	-2	-2
19	Div 01	2 / 1 = 2	2	/	1	2
20	Div 02	2 / -2 != 42	2	/	-2	42
21						

Figure 4.2: Test data for data-driven Calculator pilot

```
import sys
from TestDataParser import DataDrivenParser
from TestLibrary import CalculatorLibrary
import TestLog

# Check that name of the data file is given as an argument
if len(sys.argv) != 2:
    print "Give name of the test data file as an argument"
    sys.exit(1)

# Parse test data and get test cases
dataFile = sys.argv[1]
testData = DataDrivenParser.parse(dataFile)
testCases = testData.getTestCases()

# Initialize logging, get a logger and log a starting message
logFile = testData.getSetting("Log File")
logLevel = testData.getSetting("Log Level")
TestLog.initialize(logFile, logLevel)
logger = TestLog.getLogger("Driver")
logger.info("Executing test cases from file %s" % (dataFile))

# Variables to hold statistics
passed = executed = 0
total = len(testCases)

### Main test execution loop
for tc in testCases:
    executed += 1
    info = "%s (%s)" % (tc.getName(), tc.getDoc())
    logger.info('%s : Starting : %d/%d' % (info, executed, total))

    # Get test data from the test case container
    number1 = tc.getData("Number 1")
    number2 = tc.getData("Number 2")
    operator = tc.getData("Operator")
    expected = tc.getData("Expected")
    logger.debug("Calculation: %s %s %s = %s" % (number1, operator,
                                                 number2, expected))
    # Initialize test library and use it to lauch the Calculator
    library = CalculatorLibrary(tc.getName())
    library.launchCalculator()

    # Run test. If no exception is raised the test passes.
    try:
        library.input(number1)
        library.click(operator)
        library.input(number2)
        library.click("=")
        library.check(expected)
        logger.passed(info)
        passed += 1
    # AssertionErrors are raised if verifications fail in tests
    except AssertionError, ae:
        logger.failed("%s : Verification failed : %s" % (info, ae))
    # Other exceptions mean unexpected errors in test execution
    except Exception, e:
        logger.failed("%s : Error in test execution : %s" % (info, e))

    library.closeCalculator()

# Finally log and print summary of the test execution
summary = "%d/%d tests run, %d/%d passed, %d/%d failed" % \
          (executed, total, passed, executed, executed-passed, executed)
logger.info(summary)
print summary
print "More detailed test log in %s" % (logFile)
```

Listing 4.1: Driver script for data-driven Calculator pilot

```
import win32api, win32com.client, win32clipboard
import TestLog


class CalculatorLibrary::

    # Initialize the test library
    def __init__(self, name):
        self.logger = TestLog.getLogger(name)
        self.shell = win32com.client.Dispatch("WScript.Shell")

    # Launch the calculator - run before actual test execution
    def launchCalculator(self):
        self.shell.Run("calc", 1)
        self.logger.debug("Calculator started")
        self.delay()

    # Close the calculator after tests by sending it Alt-F4
    def closeCalculator(self):
        self.shell.AppActivate("Calculator")
        self.shell.SendKeys("%{F4}")
        self.logger.debug("Calculator closed")

    # Click Calculator's buttons
    def click(self, key):
        self.shell.AppActivate("Calculator")
        # Escape SendKeys special characters with { and }
        if key in [ "+", "^", "%", "~" ]:
            key = "{" + key + "}"
        self.shell.SendKeys(key)
        self.logger.debug("Clicked key '%s'" % (key))
        self.delay()

    # Input numbers to Calculator
    def input(self, num):
        negative = False
        # Handle negative numbers
        if num[0] == "-":
            negative = True
            num = num[1:]
        for n in num:
            self.click(n)
        if negative:
            self.click("{F9}")    # F9 is the '+/-' key

    # Check that the result is expected
    def check(self, expected):
        self.shell.AppActivate("Calculator")
        # Copy result to clipboard with ctrl-c
        self.click("^c")
        self.delay()
        # Get result from clipboard
        win32clipboard.OpenClipboard()
        actual = win32clipboard.GetClipboardData()
        win32clipboard.CloseClipboard()
        self.logger.debug("Expected: '%s', Actual: '%s'" % (expected, actual))
        if expected != actual:
            raise AssertionError, "Checking results failed: " \
                  + expected + " != " + actual

    # Give the SUT time to react after interactions
    def delay(self, ms=100):
        win32api.Sleep(ms)
```

Listing 4.2: Test library for data-driven Calculator pilot

```
Sat Jul 16 21:34:31 2005 | Driver   | INFO  | Executing test cases from file TestData.sxc
Sat Jul 16 21:34:31 2005 | Driver   | INFO  | Add 01 (1 + 2 = 3) : Starting : 1/8
Sat Jul 16 21:34:32 2005 | Driver   | PASS  | Add 01 (1 + 2 = 3)
Sat Jul 16 21:34:32 2005 | Driver   | INFO  | Add 02 (1 + -2 = -1) : Starting : 2/8
Sat Jul 16 21:34:33 2005 | Driver   | PASS  | Add 02 (1 + -2 = -1)
Sat Jul 16 21:34:33 2005 | Driver   | INFO  | Sub 01 (1 - 2 = -1) : Starting : 3/8
Sat Jul 16 21:34:34 2005 | Driver   | PASS  | Sub 01 (1 - 2 = -1)
Sat Jul 16 21:34:34 2005 | Driver   | INFO  | Sub 02 (1 - -2 = 3) : Starting : 4/8
Sat Jul 16 21:34:35 2005 | Driver   | PASS  | Sub 02 (1 - -2 = 3)
Sat Jul 16 21:34:35 2005 | Driver   | INFO  | Mul 01 (1 * 2 = 2) : Starting : 5/8
Sat Jul 16 21:34:35 2005 | Driver   | PASS  | Mul 01 (1 * 2 = 2)
Sat Jul 16 21:34:35 2005 | Driver   | INFO  | Mul 02 (1 * -2 = -2) : Starting : 6/8
Sat Jul 16 21:34:36 2005 | Driver   | PASS  | Mul 02 (1 * -2 = -2)
Sat Jul 16 21:34:36 2005 | Driver   | INFO  | Div 01 (2 / 1 = 2) : Starting : 7/8
Sat Jul 16 21:34:37 2005 | Driver   | PASS  | Div 01 (2 / 1 = 2)
Sat Jul 16 21:34:37 2005 | Driver   | INFO  | Div 02 (2 / -2 != 42) : Starting : 8/8
Sat Jul 16 21:34:38 2005 | Driver   | FAIL  | Div 02 (2 / -2 != 42) : Verification failed :
                                              42 != -1
Sat Jul 16 21:34:38 2005 | Driver   | INFO  | 8/8 tests run, 7/8 passed, 1/8 failed
```

Figure 4.3: Test log from data-driven Calculator pilot

4.3.3 Test Library

The test library which handles interacting with the tested system is shown in Listing 4.1. Calculator is used by sending it keyboard events using `SendKeys` method from the Windows Script Host, which can be accessed from a Python code using Python's Win32 modules. Checking the result is implemented so that first the result is send to clipboard (using ctrl-c), then clipboard contents are get with Win32 modules and finally expected and actual results are compared and `AssertionError` raised if comparison fails.

4.3.4 Test Log

The driver script is run from the command line and the name of the test data file is given it as an argument. After tests have been executed a summary is printed to screen along with the name of the created test log. The test log contains full test execution trace and the level of detail in the log depends on the specified logging level. The test log created in this pilot, using logging level info, is seen in Figure 4.3.

4.3.5 Summary

Implementing the driver script and the test library for data-driven Calculator pilot was a fairly easy task. That can be easily seen from the presented code and also from Table 4.3 which lists component sizes in lines of code (empty and comment lines excluded again). In total all the code for data-driven Calculator testing is less than hundred lines which is not bad at all. The current implementation is of course limited only to certain kinds of tests but writing similar driver scripts and adding new functionality to the test library is easy, especially when a base implementation is available. It would thus be possible to test Calculator, or some other Windows application, thoroughly with the framework.

My impression is that the data-driven side of the presented framework concept works very well. More experience from the framework—especially from its keyword-driven capabilities—is of course needed before the framework can be really evaluated.

Driver Script	51
Test Library	42
TOTAL	93

Table 4.3: Components sizes in data-driven Calculator pilot

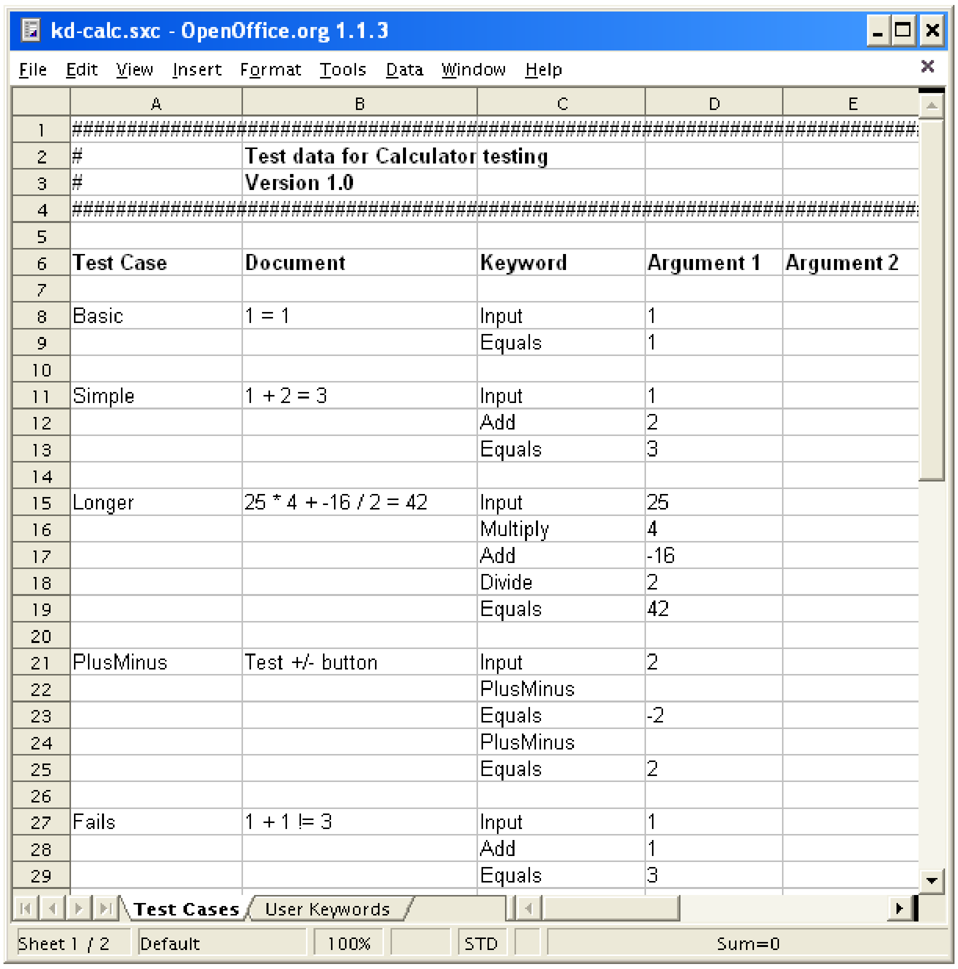
4.4 Keyword-Driven Windows Application Testing

4.4.1 Test Data

Test cases created in this pilot is shown are Figure 4.4 and constructed user keywords in Figure 4.5. Overall the test data is very similar to earlier keyword-driven testing examples in the previous chapter but has more keywords and different kinds of tests.

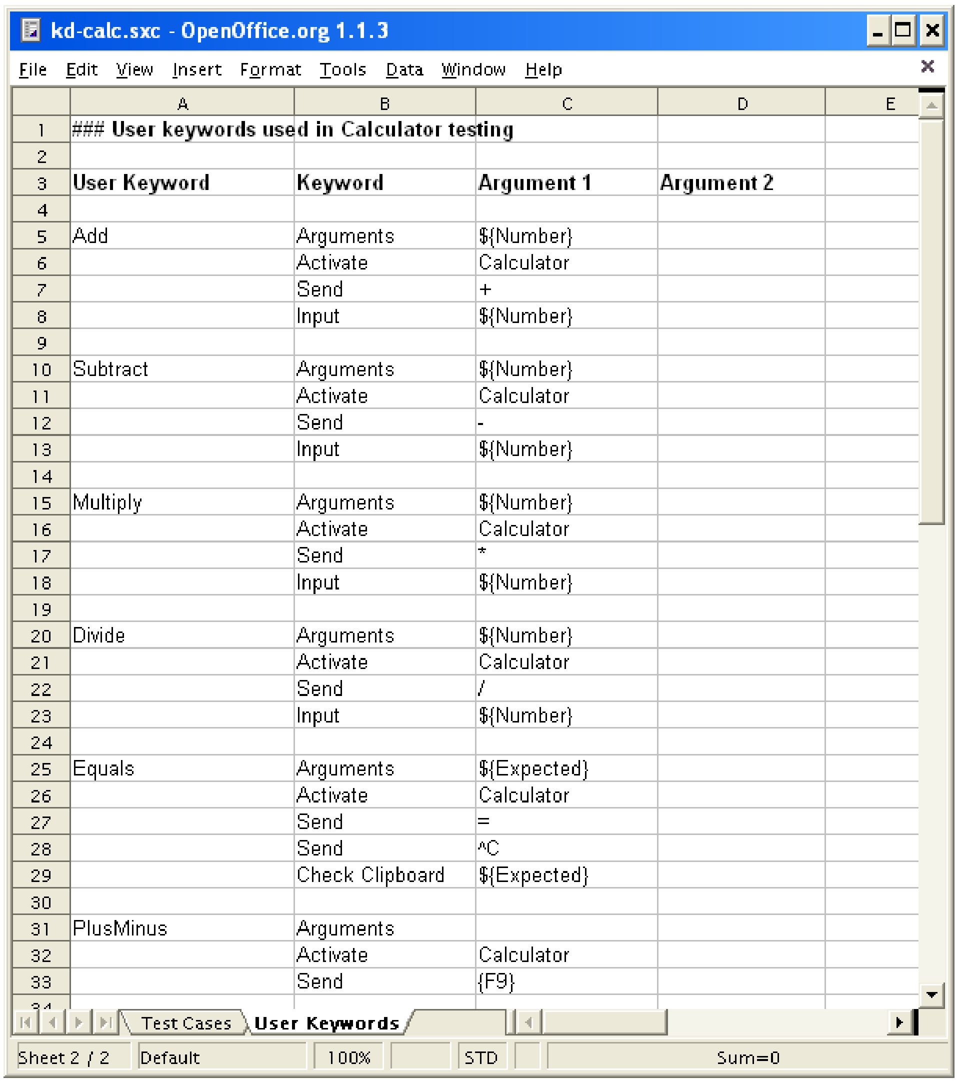
4.4.2 Driver Script

The driver script used to control the keyword-driven test execution, shown in Listing 4.3, looks pretty familiar after the data-driven driver script. Many tasks, for example parsing the test data and collecting statistics, are implemented similarly but actual test execution is naturally different. The difference is that in the data-driven version the driver script has hard coded list of actions to take (input-click-input-click-check) but now actions are specified as keywords in the test data. The driver-scripts runs specified keywords using `executeKeyword` method (copied from Listing 3.4) and if they all execute without problems the test case passes. The test status is communicated using exceptions similarly as in the data-driven version.


kd-calc.sxc - OpenOffice.org 1.1.3
File Edit View Insert Format Tools Data Window Help
Test Cases
User Keywords
Sheet 1 / 2
Default
100%
STD
Sum=0

	A	B	C	D	E
1	##				
2	#	**Test data for Calculator testing**			
3	#	**Version 1.0**			
4	##				
5					
6	**Test Case**	**Document**	**Keyword**	**Argument 1**	**Argument 2**
7					
8	Basic	1 = 1	Input	1	
9			Equals	1	
10					
11	Simple	1 + 2 = 3	Input	1	
12			Add	2	
13			Equals	3	
14					
15	Longer	25 * 4 + -16 / 2 = 42	Input	25	
16			Multiply	4	
17			Add	-16	
18			Divide	2	
19			Equals	42	
20					
21	PlusMinus	Test +/- button	Input	2	
22			PlusMinus		
23			Equals	-2	
24			PlusMinus		
25			Equals	2	
26					
27	Fails	1 + 1 != 3	Input	1	
28			Add	1	
29			Equals	3	

Figure 4.4: Test cases for keyword-driven Calculator pilot


kd-calc.sxc - OpenOffice.org 1.1.3
File Edit View Insert Format Tools Data Window Help
Test Cases
User Keywords
Sheet 2 / 2
Default
100%
STD
Sum=0

	A	B	C	D	E
1	### User keywords used in Calculator testing				
2					
3	User Keyword	Keyword	Argument 1	Argument 2	
4					
5	Add	Arguments	${Number}		
6		Activate	Calculator		
7		Send	+		
8		Input	${Number}		
9					
10	Subtract	Arguments	${Number}		
11		Activate	Calculator		
12		Send	-		
13		Input	${Number}		
14					
15	Multiply	Arguments	${Number}		
16		Activate	Calculator		
17		Send	*		
18		Input	${Number}		
19					
20	Divide	Arguments	${Number}		
21		Activate	Calculator		
22		Send	/		
23		Input	${Number}		
24					
25	Equals	Arguments	${Expected}		
26		Activate	Calculator		
27		Send	=		
28		Send	^C		
29		Check Clipboard	${Expected}		
30					
31	PlusMinus	Arguments			
32		Activate	Calculator		
33		Send	{F9}		

Figure 4.5: User keywords for keyword-driven Calculator pilot

```
import sys
from TestDataParser import KeywordDrivenParser
from CalculatorLibrary import CalculatorLibrary
import TestLog

if len(sys.argv) != 2:
    print "Give name of the test data file as an argument"
    sys.exit(1)

dataFile = sys.argv[1]
testData = KeywordDrivenParser.parse(dataFile)
testCases = testData.getTestCases()

logFile = testData.getSetting('Log File', 'Calculator.log')
logLevel = testData.getSetting('Log Level', 'debug')
TestLog.initialize(logFile, logLevel)
logger = TestLog.getLogger('Driver')
logger.info('Executing test cases from file %s' % (dataFile))

passed = 0
executed = 0
total = len(testCases)

def executeKeyword(keyword, library):
    name = keyword.getName()
    if keyword.getType() == 'user'
        logger.debug('Executing user keyword %s' % name)
        for subKeyword in keyword.getKeywords():
            executeKeyword(keyword, library)
    else:
        logger.debug('Executing base keyword %s' % name)
        handlerName = name.lower().replace(' ', '') + 'Handler'
        if not hasattr(library, handlerName):
            raise Exception, 'No handler for keyword %s' % name
        handler = getattr(library, handlerName)
        args = keyword.getArgs()
        handler(*args)

### Main test execution loop
for tc in testCases:
    executed += 1
    info = '%s (%s)' % (tc.getName(), tc.getDoc())
    logger.info('%s : Starting : %d/%d' % (info, executed, total))
    library = CalculatorLibrary(tc.getName())
    library.setUp()
    try:
        for keyword in tc.getKeywords():
            executeKeyword(keyword, library)
        logger.passed(info)
        passed += 1
    except AssertionError, err:
        logger.failed('%s : Verification failed : %s' % (info, err))
    except Exception, err:
        logger.failed('%s : Error in test execution : %s' % (info, err))
    library.tearDown()

summary = '%d/%d tests run, %d/%d passed, %d/%d failed' % \
          (executed, total, passed, executed, executed-passed, executed)
logger.info(summary)
print summary
print 'More detailed test log in %s' % (logFile)
```

Listing 4.3: Driver script for keyword-driven Calculator pilot

4.4.3 Test Library

This pilot demonstrates how the framework can be used to test Windows applications in keyword-driven style. That is why I wanted to implement a reusable test library for Windows testing and not include any Calculator specific features in it. The Windows library is seen in Listing 4.4. It should of course have more functionality to be generally reusable but in pilot use no more features were needed.

Calculator specific features were then put into a Calculator test library shown in Listing 4.5. It extends the `WindowsLibrary` so it has all the functionality implemented there but it also adds some Calculator specific functionality.

The two test libraries contain pretty much the same functionality as the test library used earlier in data-driven testing. Also the interaction with the tested system is implemented similarly using `SendKeys` methods.

Keyword-driven test libraries provide their functionality mainly through handlers. Only non-handler methods called externally are `setUp` and `tearDown`, which are used to put the test environment to a known state before and after each test case.

One more thing to note in the Windows library is that most of its handlers do not have much functionality themselves but instead they use internal helper methods. An example of this is the `launchHandler` which uses `launch` method. This arrangement helps reusing code as the helper methods can be called by other methods also. For example Calculator library's `setUp` and `tearDown` methods use them to launch and close Calculator.

4.4.4 Test Log

Test logs were created similarly in keyword-driven and data-driven pilots. Thus the test log in Figure 4.6 is also nearly identical to the earlier test log. To make logs a bit different the logging level is now debug.

4.4.5 Summary

Testing Calculator with the keyword-driven approach proved out to be at least as easy as testing it with the data-driven style. This is actually no surprise as all the required framework components were ready for both. Comparing component sizes in Table 4.3 and Table 4.4 reveals that keyword-driven implementation is slightly bigger (105 vs. 89 lines) but this difference is insignificant. An interesting thing to

```
import win32api, win32com.client, win32clipboard


class WindowsLibrary:

    def __init__(self, name):
        BaseTestLibrary.__init__(self, name)
        self.shell = win32com.client.Dispatch("WScript.Shell")

    ### Keyword handlers

    def launchHandler(self, app):
        self.launch(app)

    def sendHandler(self, keys):
        self.send(keys)

    def activateHandler(self, app):
        self.activate(app)

    def checkclipboardHandler(self, expected):
        win32clipboard.OpenClipboard()
        actual = win32clipboard.GetClipboardData()
        win32clipboard.CloseClipboard()
        if expected != actual:
            raise AssertionError, "Clipboard verification failed: " \
                  + expected + " != " + actual
        self.logger.debug("Clipboard verification succeeded (%s)" % actual)

    def delayHandler(self, seconds):
        seconds = int(seconds.replace(",", "."))
        self.delay(seconds * 1000)

    ### Helper functions

    def launch(self, app):
        rc = self.shell.Run(app, 1)
        if rc != 0:
            self.fail("Launcing application '%s' failed with RC %d" % (app, rc))
        self.logger.trace("Application '%s' launched" % (app))
        self.delay()

    def send(self, keys):
        if keys in [ "+", "^", "%", "~" ]:
            keys = "{" + keys + "}"
        self.shell.SendKeys(keys)
        self.logger.trace("Key(s) '%s' sent" % (keys))
        self.delay()

    def activate(self, app):
        status = self.shell.AppActivate(app)
        if status != True:
            self.fail("Activating application '%s' failed" % (app))
        self.logger.trace("App '%s' activated" % (app))

    def delay(self, ms=100):
        self.logger.trace("Sleeping %d ms" % (ms))
        win32api.Sleep(ms)
```

Listing 4.4: Generic Windows test library

```
from WindowsLibrary import WindowsLibrary


class CalculatorLibrary(WindowsLibrary):

    def __init__(self, name):
        WindowsLibrary.__init__(self, name)

    def setUp(self):
        self.launch("calc")
        self.delay()
        self.logger.trace("Set up done")

    def tearDown(self):
        self.activate("Calculator")
        self.delay()
        self.send("%{F4}")
        self.logger.trace("Tear down done")

    def inputHandler(self, nuumber):
        if number[0] == "-":
            self.send(number[1:])
            self.send("{F9}")      # F9 is the '+/-' key
        else:
            self.send(num)
```

Listing 4.5: Calculator specific test library

```
Sat Jul 16 20:53:06 2005 | Driver    | INFO  | Executing test cases from file TestData.sxc
Sat Jul 16 20:53:06 2005 | Driver    | INFO  | Basic (1 = 1) : Starting : 1/5
Sat Jul 16 20:53:07 2005 | Basic     | DEBUG | Clipboard verification succeeded (1)
Sat Jul 16 20:53:07 2005 | Driver    | PASS  | Basic (1 = 1)
Sat Jul 16 20:53:07 2005 | Driver    | INFO  | Simple (1 + 2 = 3) : Starting : 2/5
Sat Jul 16 20:53:08 2005 | Simple    | DEBUG | Clipboard verification succeeded (3)
Sat Jul 16 20:53:08 2005 | Driver    | PASS  | Simple (1 + 2 = 3)
Sat Jul 16 20:53:08 2005 | Driver    | INFO  | Longer (25 * 4 + -16 / 2 = 42) : Starting :
                                               3/5
Sat Jul 16 20:53:09 2005 | Longer    | DEBUG | Clipboard verification succeeded (42)
Sat Jul 16 20:53:09 2005 | Driver    | PASS  | Longer (25 * 4 + -16 / 2 = 42)
Sat Jul 16 20:53:09 2005 | Driver    | INFO  | PlusMinus (Test +/- button) : Starting :
                                               4/5
Sat Jul 16 20:53:10 2005 | PlusMinus | DEBUG | Clipboard verification succeeded (-2)
Sat Jul 16 20:53:10 2005 | PlusMinus | DEBUG | Clipboard verification succeeded (2)
Sat Jul 16 20:53:10 2005 | Driver    | PASS  | PlusMinus (Test +/- button)
Sat Jul 16 20:53:10 2005 | Driver    | INFO  | Fails (1 + 1 != 3) : Starting : 5/5
Sat Jul 16 20:53:11 2005 | Driver    | FAIL  | Fails (1 + 1 != 3) : Verification failed :
                                               3 != 2
Sat Jul 16 20:53:11 2005 | Driver    | INFO  | 5/5 tests run, 4/5 passed, 1/5 failed
```

Figure 4.6: Test log from keyword-driven Calculator pilot

notice is how small set of base keywords is needed when higher level keywords can be created as user keywords. For example calculations (Add, Multiply, etc.) need no additional code in the test library.

This pilot also proves how flexible keyword-driven testing is compared to data-driven testing. This difference has been pointed out several times before but it can be truly understood only after seeing it in real situation testing the same application. In the data-driven pilot's summary I wrote that thorough testing of Calculator would only require few more driver scripts and a bit of new functionality in the test library. It is a huge difference to be able to conclude that thorough testing could be achieved simply by designing more test cases.

Driver Script	57
Windows Library	41
Calculator Library	19
TOTAL	117

Table 4.4: Component sizes in keyword-driven Calculator pilot

4.5 Keyword-Driven Web Testing

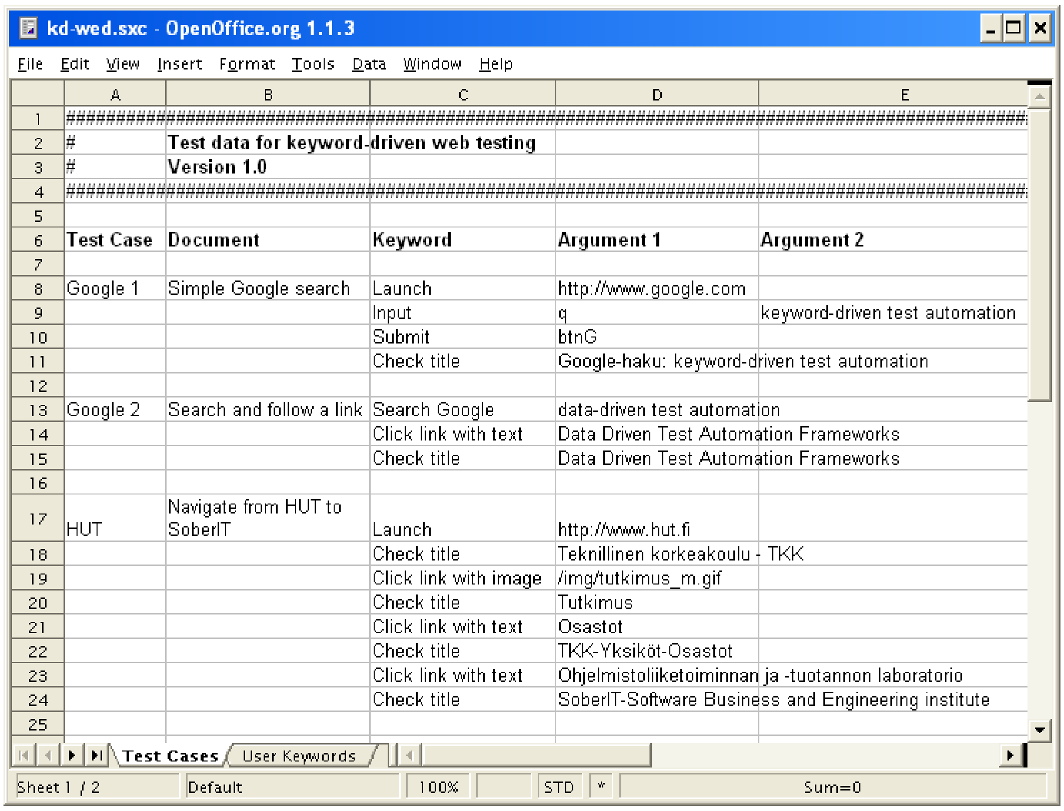
4.5.1 Test Data

The test data used in this pilot is presented in Figure 4.7. These test cases are not too good as tests—they do not really test much—but that is ok as their main task is helping to find out how well the keyword-driven approach suits for web testing.

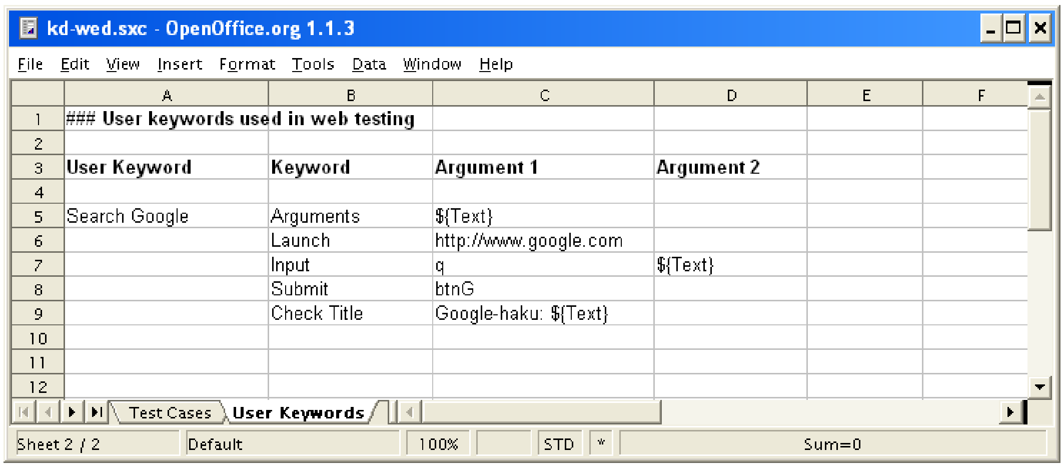
One thing to note from the test data is how user keywords make test cases shorter. Four lines in the test case Google 1 are equivalent to the one Search Google line in the test case Google 2. User keywords make it possible to have short test which actually do plenty of tasks behind the scenes.

4.5.2 Driver Script

The driver script implemented for keyword-driven web pilot is shown in Listing 4.6. It is nearly identical with the keyword-driven driver script used with Calculator. The first of the two differences is that because the test library uses jWebUnit, web


kd-wed.sxc - OpenOffice.org 1.1.3
File Edit View Insert Format Tools Data Window Help
Test Cases
User Keywords
Sheet 1 / 2
Default
100%
STD
Sum=0

	A	B	C	D	E
1	##				
2	#	Test data for keyword-driven web testing			
3	#	Version 1.0			
4	##				
5					
6	Test Case	Document	Keyword	Argument 1	Argument 2
7					
8	Google 1	Simple Google search	Launch	http://www.google.com	
9			Input	q	keyword-driven test automation
10			Submit	btnG	
11			Check title	Google-haku: keyword-driven test automation	
12					
13	Google 2	Search and follow a link	Search Google	data-driven test automation	
14			Click link with text	Data Driven Test Automation Frameworks	
15			Check title	Data Driven Test Automation Frameworks	
16					
17	HUT	Navigate from HUT to SoberIT	Launch	http://www.hut.fi	
18			Check title	Teknillinen korkeakoulu - TKK	
19			Click link with image	/img/tutkimus_m.gif	
20			Check title	Tutkimus	
21			Click link with text	Osastot	
22			Check title	TKK-Yksiköt-Osastot	
23			Click link with text	Ohjelmistoliiketoiminnan ja -tuotannon laboratorio	
24			Check title	SoberIT-Software Business and Engineering institute	
25					


kd-wed.sxc - OpenOffice.org 1.1.3
File Edit View Insert Format Tools Data Window Help
Test Cases
User Keywords
Sheet 2 / 2
Default
100%
STD
Sum=0

	A	B	C	D	E	F
1	### User keywords used in web testing					
2						
3	User Keyword	Keyword	Argument 1	Argument 2		
4						
5	Search Google	Arguments	${Text}			
6		Launch	http://www.google.com			
7		Input	q	${Text}		
8		Submit	btnG			
9		Check Title	Google-haku: ${Text}			
10						
11						
12						

Figure 4.7: Test cases and user keywords for keyword-driven web pilot

testing tool implemented with Java and using JUnit internally, JUnit's assertions and all possible Java exceptions needed to be caught. The second difference is that this time no set up or tear down is needed and the driver script does not need to call library methods implementing them.

Similarities in two different driver scripts raise interesting questions about being able to make them more generic. A trivial change into better direction is taking `executeKeyword` method out and placing it into a separate reusable module but similarities do not end there. Only real differences between the two drivers are test libraries they use and different ways to set up the test environment and also these tasks could be done in a more generic manner. How that is possible and what benefits it would bring are discussed further when pilot results are evaluated in the next chapter.

4.5.3 Test Library

The test library handling actual web testing is shown in Listing 4.7. The biggest task in implementing it was finding a suitable web testing tool. After evaluating few alternatives I selected jWebUnit, which is a wrapper for better known HTTPUnit, which in turn is build on top of JUnit. The fact that jWebUnit is implemented with Java was not a problem because the framework had been designed to be Jython compatible from the beginning.

As it can be seen from the code, the actual implementation of the test library was a very simple task. That is mainly due to jWebUnit being a really good tool and extremely easy to embed. The whole test library is basically just a thin wrapper around jWebUnit's WebTester class.

4.5.4 Test Log

The test log produced by this pilot is seen in Figure 4.8. This time the logging level is trace, lowest possible, so that all logging from various components is shown. Going through the log gives a good insight of how the framework works.

4.5.5 Summary

As I was equipped with the experience from previous pilots, implementing needed components for keyword-driven web pilot was very easy. I could reuse the previous

```
import sys
import java.lang.Exception
import junit.framework.AssertionFailedError
from TestDataParser import KeywordDrivenParser
from WebLibrary import WebLibrary
import TestLog

if len(sys.argv) != 2:
    print "Give name of the test data file as an argument"
    sys.exit(1)

dataFile = sys.argv[1]
testData = KeywordDrivenParser.parse(dataFile)
testCases = testData.getTestCases()

logFile = testData.getSetting("Log File", "WebTest.log")
logLevel = testData.getSetting("Log Level", "trace")
TestLog.initialize(logFile, logLevel)
logger = TestLog.getLogger("Driver")
logger.info("Executing test cases from file %s" % (dataFile))

passed = executed = 0
total = len(testCases)

def executeKeyword(keyword, library):
    name = keyword.getName()
    if keyword.getType() == 'user'
        logger.debug('Executing user keyword %s' % name)
        for subKeyword in keyword.getKeywords():
            executeKeyword(keyword, library)
    else:
        logger.debug('Executing base keyword %s' % name)
        handlerName = name.lower().replace(' ', '') + 'Handler'
        if not hasattr(library, handlerName):
            raise Exception, 'No handler for keyword %s' % name
        handler = getattr(library, handlerName)
        args = keyword.getArgs()
        handler(*args)

### Main test execution loop
for tc in testCases:
    executed += 1
    info = "%s (%s)" % (tc.getName(), tc.getDoc())
    logger.info('%s : Starting : %d/%d' % (info, executed, total))
    library = WebLibrary(tc.getName())
    try:
        for keyword in tc.getKeywords():
            executeKeyword(keyword, library)
        logger.passed(info)
        passed += 1
    # Both PyUnit and JUnit based assertions are failures
    except (AssertionError, junit.framework.AssertionFailedError), err:
        logger.failed("%s : Verification failed : %s" % (info, err))
    # Rest exceptions, both Python and Java based, are errors
    except (Exception, java.lang.Exception), err:
        logger.failed("%s : Error in test execution : %s" % (info, err))

summary = "%d/%d tests run, %d/%d passed, %d/%d failed" % \
          (executed, total, passed, executed, executed-passed, executed)
logger.info(summary)
print summary
print "More detailed test log in %s" % (logFile)
```

Listing 4.6: Driver script for keyword-driven web pilot

```
from net.sourceforge.jwebunit import WebTester

class WebLibrary:

    def __init__(self, name):
        BaseTestLibrary.__init__(self, name)
        self.tester = WebTester()

    def launchHandler(self, url):
        self.tester.getTestContext().setBaseUrl(url)
        self.tester.beginAt("/")
        self.logger.trace("Launched '%s'" % (url))

    def checktitleHandler(self, expected):
        self.tester.assertTitleEquals(expected)
        self.logger.debug("Verifying title succeeded (%s)" % (expected))

    def inputHandler(self, fieldId, text):
        self.tester.setFormElement(fieldId, text)
        self.logger.trace("Text '%s' written to '%s'" % (text, fieldId))

    def clickbuttonHandler(self, buttonId):
        self.tester.clickButton(buttonId)
        self.logger.trace("Clicked button '%s'" % (buttonId))

    def submitHandler(self, buttonId):
        self.tester.submit(buttonId)
        self.logger.trace("Submitted '%s'" % (buttonId))

    def clicklinkHandler(self, linkId):
        self.tester.clickLink(linkId)
        self.logger.trace("Clicked link '%s'" % (linkId))

    def clicklinkwithtextHandler(self, text):
        self.tester.clickLinkWithText(text)
        self.logger.trace("Clicked link with text '%s'" % (text))

    def clicklinkwithimageHandler(self, src):
        self.tester.clickLinkWithImage(src)
        self.logger.trace("Clicked link with image '%s'" % (src))
```

Listing 4.7: Test library for keyword-driven web pilot

```
2005-07-17 02:44:09 | Driver   | INFO  | Executing test cases from file TestData.sxc
2005-07-17 02:44:09 | Driver   | INFO  | Google 1 (Simple Google search) : Starting : 1/3
2005-07-17 02:44:09 | Google 1 | TRACE | Executing framework keyword 'Launch'
2005-07-17 02:44:11 | Google 1 | TRACE | Launched 'http://www.google.com'
2005-07-17 02:44:12 | Google 1 | TRACE | Keyword 'Launch' executed
2005-07-17 02:44:12 | Google 1 | TRACE | Executing framework keyword 'Input'
2005-07-17 02:44:12 | Google 1 | TRACE | Text 'keyword-driven test automation' written to 'q'
2005-07-17 02:44:12 | Google 1 | TRACE | Keyword 'Input' executed
2005-07-17 02:44:12 | Google 1 | TRACE | Executing framework keyword 'Submit'
2005-07-17 02:44:12 | Google 1 | TRACE | Submitted 'btnG'
2005-07-17 02:44:12 | Google 1 | TRACE | Keyword 'Submit' executed
2005-07-17 02:44:13 | Google 1 | TRACE | Executing framework keyword 'Check Title'
2005-07-17 02:44:13 | Google 1 | DEBUG | Verifying title succeeded (Google-haku: keyword-d...
2005-07-17 02:44:13 | Google 1 | TRACE | Keyword 'Check Title' executed
2005-07-17 02:44:13 | Driver   | PASS  | Google 1 (Simple Google search)
2005-07-17 02:44:13 | Driver   | INFO  | Google 2 (Search and follow a link) : Starting : 2/3
2005-07-17 02:44:13 | Google 2 | TRACE | Executing user keyword 'Search Google'
2005-07-17 02:44:13 | Google 2 | TRACE | Executing framework keyword 'Launch'
2005-07-17 02:44:14 | Google 2 | TRACE | Launched 'http://www.google.com'
2005-07-17 02:44:14 | Google 2 | TRACE | Keyword 'Launch' executed
2005-07-17 02:44:14 | Google 2 | TRACE | Executing framework keyword 'Input'
2005-07-17 02:44:14 | Google 2 | TRACE | Text 'data-driven test automation' written to 'q'
2005-07-17 02:44:14 | Google 2 | TRACE | Keyword 'Input' executed
2005-07-17 02:44:14 | Google 2 | TRACE | Executing framework keyword 'Submit'
2005-07-17 02:44:15 | Google 2 | TRACE | Submitted 'btnG'
2005-07-17 02:44:15 | Google 2 | TRACE | Keyword 'Submit' executed
2005-07-17 02:44:15 | Google 2 | TRACE | Executing framework keyword 'Check Title'
2005-07-17 02:44:15 | Google 2 | DEBUG | Verifying title succeeded (Google-haku: data-driv...
2005-07-17 02:44:15 | Google 2 | TRACE | Keyword 'Check Title' executed
2005-07-17 02:44:15 | Google 2 | TRACE | Keyword 'Search Google' executed
2005-07-17 02:44:15 | Google 2 | TRACE | Executing framework keyword 'Click Link With Text'
2005-07-17 02:44:21 | Google 2 | TRACE | Clicked link with text 'Data Driven Test Automati...
2005-07-17 02:44:21 | Google 2 | TRACE | Keyword 'Click Link With Text' executed
2005-07-17 02:44:22 | Google 2 | TRACE | Executing framework keyword 'Check Title'
2005-07-17 02:44:22 | Google 2 | DEBUG | Verifying title succeeded (Data Driven Test Autot...
2005-07-17 02:44:22 | Google 2 | TRACE | Keyword 'Check Title' executed
2005-07-17 02:44:22 | Driver   | PASS  | Google 2 (Search and follow a link)
2005-07-17 02:44:22 | Driver   | INFO  | HUT (Navigate from HUT to SoberIT) : Starting : 3/3
2005-07-17 02:44:22 | HUT      | TRACE | Executing framework keyword 'Launch'
2005-07-17 02:44:23 | HUT      | TRACE | Launched 'http://www.hut.fi'
2005-07-17 02:44:23 | HUT      | TRACE | Keyword 'Launch' executed
2005-07-17 02:44:23 | HUT      | TRACE | Executing framework keyword 'Check Title'
2005-07-17 02:44:23 | HUT      | DEBUG | Verifying title succeeded (Teknillinen korkeakoul...
2005-07-17 02:44:23 | HUT      | TRACE | Keyword 'Check Title' executed
2005-07-17 02:44:23 | HUT      | TRACE | Executing framework keyword 'Click Link With Image'
2005-07-17 02:44:23 | HUT      | TRACE | Clicked link with image '/img/tutkimus_m.gif'
2005-07-17 02:44:23 | HUT      | TRACE | Keyword 'Click Link With Image' executed
2005-07-17 02:44:23 | HUT      | TRACE | Executing framework keyword 'Check Title'
2005-07-17 02:44:23 | HUT      | DEBUG | Verifying title succeeded (Tutkimus)
2005-07-17 02:44:23 | HUT      | TRACE | Keyword 'Check Title' executed
2005-07-17 02:44:23 | HUT      | TRACE | Executing framework keyword 'Click Link With Text'
2005-07-17 02:44:24 | HUT      | TRACE | Clicked link with text 'Osastot'
2005-07-17 02:44:24 | HUT      | TRACE | Keyword 'Click Link With Text' executed
2005-07-17 02:44:24 | HUT      | TRACE | Executing framework keyword 'Check Title'
2005-07-17 02:44:24 | HUT      | DEBUG | Verifying title succeeded (TKK-Yksiköt-Osastot)
2005-07-17 02:44:24 | HUT      | TRACE | Keyword 'Check Title' executed
2005-07-17 02:44:24 | HUT      | TRACE | Executing framework keyword 'Click Link With Text'
2005-07-17 02:44:25 | HUT      | TRACE | Clicked link with text 'Ohjelmistoliiketoiminnan ...
2005-07-17 02:44:25 | HUT      | TRACE | Keyword 'Click Link With Text' executed
2005-07-17 02:44:25 | HUT      | TRACE | Executing framework keyword 'Check Title'
2005-07-17 02:44:25 | HUT      | DEBUG | Verifying title succeeded (SoberIT-Software Busin...
2005-07-17 02:44:25 | HUT      | TRACE | Keyword 'Check Title' executed
2005-07-17 02:44:25 | Driver   | PASS  | HUT (Navigate from HUT to SoberIT)
2005-07-17 02:44:25 | Driver   | INFO  | 3/3 tests run, 3/3 passed, 0/3 failed
```

Figure 4.8: Test log from keyword-driven web pilot

driver script and jWebUnit made implementing the test library trivial. The amount of new code is thus much more smaller than Table 4.5 tells.

An interesting finding was that the driver scripts used in different keyword-driven pilots were very similar and they could easily be made more generic. If it was possible to have only one generic driver script it would fasten implementation and significantly decrease the amount of code to maintain. This important issue is discussed more thoroughly in next chapter when results of these pilots are evaluated.

Driver Script	61
Web Library	30
TOTAL	91

Table 4.5: Component sizes in keyword-driven wed testing

4.6 Chapter Summary

Implementing needed framework and test specific components to the three pilots succeeded well. The hardest task was designing and implementing the parser components and after they were ready everything else was relatively easy. Most notable, writing test libraries turned out to be very simple. Table 4.6 lists sizes off all implemented components and reveals that total amount of code was clearly below one thousand lines.

Next chapter will continue from here by collecting pilot experiences together. Based on the results the overall feasibility of the framework can be evaluated and possible changes suggested.

Test Data Parser	479
Logger	82
Data-Driven Calculator Testing	93
Keyword-Driven Calculator Testing	117
Keyword-Driven Web Testing	91
TOTAL	862

Table 4.6: Sizes of all components implemented in the pilot

Chapter 5

Results

In this chapter it is time to evaluate the feasibility of the framework concept that has been presented in Chapter 3 based on the experiences from the pilot in Chapter 4. Another objective is adapting the requirement set that has been defined in Chapter 2 based on the new knowledge gained in the pilot.

5.1 Feasibility of the Framework Concept

The first implementation task of the pilot was creating prototypes of the reusable framework components (Section 4.2). The main outcome was a parser that reads data from OpenOffice spreadsheet files and is capable to process both data-driven and keyword-driven test data. In addition to that also simple logger was implemented. Few internal design changes and rewrites were needed but otherwise implementation succeeded without any bigger problems.

Next tasks were implementing automated tests for Windows Calculator using both the data-driven (Section 4.3) and keyword-driven approaches (Section 4.4) and using latter also for web testing (Section 4.5). Implementing needed driver scripts and test libraries proceeded without any surprises and in the end automated tests could be executed for all tested systems. Interestingly only about hundred lines of code per one tested system was needed.

The overall feasibility of the framework is evaluated in Table 5.1 against the three high level requirements presented originally in Table 2.1. The evaluation is of course based on a quite small pilot and it also likely to be biased since it is done by the same person who has designed and implemented the framework itself. It is certain,

Automatic Test Execution	The prototype implemented had only those features that were absolutely needed but even with them test execution, analyzing the outcome and reporting results could be automated fairly easily.
Ease of Use	Usability is always a subjective matter but creating test cases with the framework proved to be easy both for me and also for those people I introduced the prototype. Even creating user keywords was considered fairly straightforward.
Maintainability	The prototype framework itself was surprisingly small which suggests that it ought to be pretty easily maintainable. For anyone using the framework that is not the important issue since they see it mainly as a black box. The much more important thing is that the test libraries and driver scripts required for each pilot were both very small and relatively easy to read which makes them easy to maintain. Based on the pilots it also seemed that separating the test data from the automation code really eased maintainability of both the code and test cases constructed in the data.

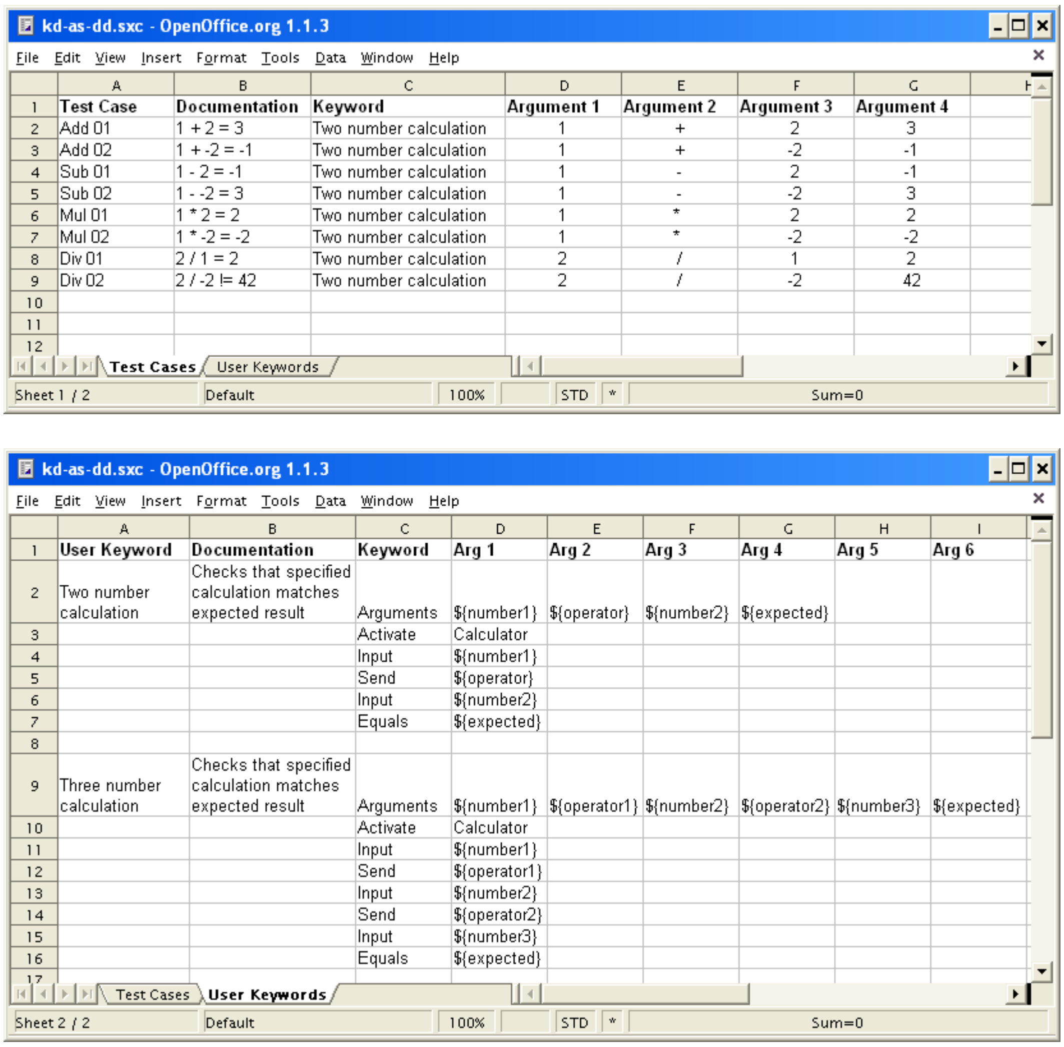
Table 5.1: Evaluating the framework against high level requirements

however, that the framework works well at least in similar contexts as in the pilot. Proving that it works in general in system and component level acceptance testing would require a larger study but there does not seem to be any problems preventing it. Based on the positive experiences and good results it can be asserted that the framework concept presented in this thesis is valid.

5.2 Changes to the Framework and Requirements

5.2.1 Using Only Keyword-Driven Approach

While implementing the prototype framework and using it in pilots I started to like keyword-driven technique more and more. The only place where I thought data-driven technique performed better was presenting multiple similar test cases with differences only in the data. For example test cases created in the data-driven pilot are presented a lot more compact way in Figure 4.2 than corresponding keyword-driven test cases in Figure 4.4. Of course latter format can be used for running different kinds of tests—main benefit of keyword-driven testing—but the former is


kd-as-dd.sxc - OpenOffice.org 1.1.3
File Edit View Insert Format Tools Data Window Help
Test Cases
User Keywords
Sheet 1 / 2
Default
100%
STD
Sum=0
Sheet 2 / 2

Test Case	Documentation	Keyword	Argument 1	Argument 2	Argument 3	Argument 4
Add 01	1 + 2 = 3	Two number calculation	1	+	2	3
Add 02	1 + -2 = -1	Two number calculation	1	+	-2	-1
Sub 01	1 - 2 = -1	Two number calculation	1	-	2	-1
Sub 02	1 - -2 = 3	Two number calculation	1	-	-2	3
Mul 01	1 * 2 = 2	Two number calculation	1	*	2	2
Mul 02	1 * -2 = -2	Two number calculation	1	*	-2	-2
Div 01	2 / 1 = 2	Two number calculation	2	/	1	2
Div 02	2 / -2 != 42	Two number calculation	2	/	-2	42

User Keyword	Documentation	Keyword	Arg 1	Arg 2	Arg 3	Arg 4	Arg 5	Arg 6
Two number calculation	Checks that specified calculation matches expected result	Arguments	${number1}	${operator}	${number2}	${expected}		
		Activate	Calculator					
		Input	${number1}					
		Send	${operator}					
		Input	${number2}					
		Equals	${expected}					
Three number calculation	Checks that specified calculation matches expected result	Arguments	${number1}	${operator1}	${number2}	${operator2}	${number3}	${expected}
		Activate	Calculator					
		Input	${number1}					
		Send	${operator1}					
		Input	${number2}					
		Send	${operator2}					
		Input	${number3}					
		Equals	${expected}					

Figure 5.1: Keyword-driven technique used as data-driven technique

clearly much better suited for presenting simple variations in the test data.

While I was experimenting with keyword-driven parts of the implemented framework I noticed that it was possible to achieve the same terse presentation that the data-driven approach provided with user keywords. For example test cases in Figure 4.2 can be implemented with keyword-driven approach as shown in Figure 5.1. Created user keyword Two number calculation actually does all the same testing tasks as the data-driven driver script in Listing 4.1 but it is of course a lot easier to understand and maintain. As it can be seen from the example, creating similar user keyword for a longer test is also a trivial tasks and leaves very little extra to maintain—something that cannot be said about creating a new driver script.

Based on these findings I conclude that only the keyword-driven part of the presented framework is actually needed. That is not because the data-driven technique would not work but just because the keyword-driven technique works so much better and, as just demonstrated, it can be used in a data-driven way if needed. Having only one way of working also makes learning the framework easier and smaller code base to maintain in the framework is always a plus.

I still believe pure data-driven frameworks have their uses in situations where larger frameworks are not needed or are not adequate. As we can remember from Listing 2.1, creating a simple data-driven parser is a very easy task and some kind of a framework can be easily implemented around it. When building a larger framework, like the one presented in this thesis, the added complexity of implementing a keyword-driven system is worth the effort.

5.2.2 Set Up and Tear Down

One thing limiting the flexibility of the keyword-driven pilots was that setting up the test environment needed to be handled separately. In Windows Calculator pilot (Section 4.4) it was done having special `setUp` and `tearDown` methods in the test library and calling them from the driver script. In web testing pilot (Section 4.5) no set up was needed as all test cases simply had `Launch` as their first keyword. The problem with the former method is that changing `setUp` and `tearDown` methods requires editing code which is against the idea of giving test designers free hands to construct test cases using only the test data. The latter approach works ok in this case but if same set up steps were needed for all test cases repeating them is not such a good idea. More importantly, using last keyword for cleaning the environment does not work since if the test fails it may never be executed.

These same problems exists also on unit level and there xUnit frameworks have solved them cleverly by allowing test designers to create separate set up and tear down functions for a group of test cases. These frameworks also make sure that set up and tear down are executed for each test before and after the test regardless the test status. Similarly a keyword-driven framework could let test designers specify in the test data what keywords to execute as set up and tear down. The framework would then take care of executing set up and tear down keywords and make sure that tear down is always executed. This addition would not change the high level layout of the framework but would of course require some changes to the keyword-driven parser.

5.2.3 Test Suites

Another feature from xUnit frameworks that I started to miss while working with the framework is grouping related test cases together into test suites. I realized that when the number of test cases grows large it gets important to be able run selected tests, for example all smoke or regression tests, easily. Another use for test suites is creating larger test case collections by grouping suites together. That kind of usage of course requires that suites can contain other suites and it would be convenient that suites also had their own set up and tear down functionalities.

5.2.4 Generic Driver Script

In Section 4.5.5 it was already mentioned that driver scripts for both keyword-driven pilots (Listings 4.3 and 4.6) are very similar. Only real differences between the two are test libraries they use and different ways to set up the test environment. Both of these tasks could, however, be done in a more generic manner. The possibility to set up and tear down the test environment with keywords was already discussed in Section 5.2.2. Selecting what test libraries to use is even easier than that—they could be simply specified along the test data similarly as settings like logging level.

Not needing system specific driver scripts is a very big improvement because there would be a lot less code to implement and maintain. Only things needed when automating tests for a new system would be test libraries and the test data. Since test libraries can also be reusable with similar systems (e.g. different web sites can be tested with the same web testing library) in many cases just designing new test cases would be enough. I do not believe test automation can get much easier than that.

Since driver script is not too descriptive name for a generic code running all tests I prefer calling it *a test runner* instead. This name change can be seen also in Figure 5.2 that shows a revised version of the detailed framework view originally presented in Figure 3.2. The new version also has test libraries taken out of the test execution system which points out that the core of the framework is totally generic and it communicates with test libraries over a clear, defined interface.

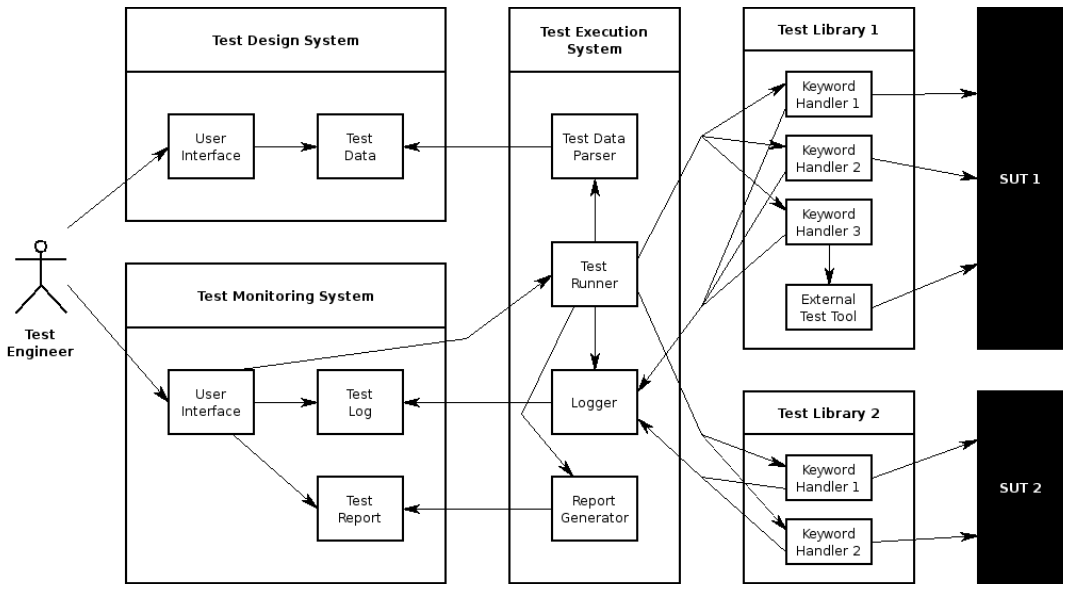


Figure 5.2: A revised detailed view of the framework

5.3 Revised List of Requirements

The changes suggested in this chapter have some effects to the requirements specified in Chapter 2. High level requirements are not affected but some of the detailed requirements need to be updated. The updated list of requirements is presented in Table 5.2.

5.4 Chapter Summary

This chapter evaluated the feasibility of the presented test automation framework concept based on the pilot experiences. The overall assessment was positive and the framework was declared valid.

Even though the framework in general functioned well some changes to it were deemed necessary. The biggest change is suggesting that using only the keyword-driven approach is enough because with user keywords it can, when needed, be used also in a data-driven manner. Other suggested changes include implementing set ups and tear downs as well as test suites, and using a generic test runner instead of system specific driver scripts. These of course affected the list of requirements specified in Chapter 2 and an adapted list is presented in Table 5.2.

In the next chapter it is time for conclusions and closing words.

High Level Requirements	The framework MUST execute test cases automatically. That includes also for example verifying results, handling errors and reporting results. The framework MUST be easy to use without programming skills. The framework MUST be easily maintainable.
Automatic Test Execution	The framework MUST be able to execute tests unattended. It MUST be possible to start and stop test execution manually. It SHOULD be possible to start test execution automatically at predefined time. It SHOULD be possible to start test execution automatically after certain events. Non-fatal errors caused by the SUT or the test environment MUST be handled gracefully without stopping test execution. Test results MUST be verified. Every executed test case MUST be assigned either Pass or Fail status and failed test cases SHOULD have a short error message explaining the failure. Framework SHOULD differentiate expected failures caused by known problems from new failures. Test execution MUST be logged. Test execution SHOULD be logged using different, configurable logging levels. Test report MUST be created automatically. Test report SHOULD be published automatically.
Ease of Use	The framework MUST use keyword-driven approach. The framework MUST support creating user keywords. The framework SHOULD support specifying common set up and tear down functionality for test cases. The framework SHOULD support grouping related test cases into test suites.
Maintainability	The framework MUST be modular. The framework SHOULD be implemented using high level scripting languages. The testware in the framework MUST be under version control. The framework MUST have coding and naming conventions. The testware MUST be adequately documented. Testability of the tested system MUST be increased as much as possible. Clear roles MUST be assigned.

Table 5.2: Revised requirements for a large scale test automation framework

Chapter 6

Conclusions

The main objective of this chapter is to look back into the requirements that were set for this thesis in the first chapter and evaluate how well they have been met. Besides that also future of the presented framework is briefly discussed before the final words.

If there is one single thing I would like a casual reader to remember from this thesis it is that test automation is a very large and non-trivial subject. As Section 1.2 points out, there are countless possible approaches and they suite differently into different contexts. While this thesis only concentrates on large scale test-automation frameworks for test execution and reporting, other approaches are also important.

The requirements for this thesis were specified in Section 1.5 and Table 6.1 lists what has been done to achieve them. As it can be seen all four requirements have been clearly fulfilled. Based on that I conclude that this thesis meets its requirements.

Writing this thesis has been a big but rewarding task. My knowledge about test automation in general and data-driven and keyword-driven frameworks in particular has increased enormously while studying all the material and implementing and using the framework prototype. That can be seen for example from the fact that after the pilots I realized that keyword-driven approach alone is a sufficient base for large scale frameworks.

After the prototype had been implemented but before finishing the thesis the framework has been rewritten in a project based on the revised requirements in Table 5.2. It has also been taken into a real use and so far the overall feedback has been mostly positive. It thus seems that the benefits promised in this thesis really are valid also in a real life project.

My hope is that some time in the near future the framework can also be released as open source in one format or another. That way ideas presented in this thesis would be easily available in a functional format for everyone and other people could extend the framework into new directions. While waiting for that to happen, however, it is time to write the final period of this thesis.

Define requirements for large scale test automation frameworks.	Chapter 2 defined a set of requirements a large scale test automation framework ought to meet. High level requirements were listed in the beginning of the chapter in Table 2.1 and included automatic test execution, ease of use and maintainability. High level requirements were broken down into set of more detailed and concrete requirements and listed in Table 2.3.
Design a framework meeting these requirements.	A framework concept was designed in Chapter 3 based on the requirements gathered in Chapter 2. Using data-driven and keyword-driven approaches were considered most important lower level requirements in terms of ease-of-use and maintainability and the presented concept naturally had these capabilities.
Test the designed framework against the defined requirements in a pilot.	The framework concept was piloted in Chapter 4 where a prototype was implemented and automated tests created for two different systems using it. In the data-driven pilot everything went pretty much as expected—the approach worked very well and proved to be useful. Keyword-driven pilots proceeded without problems as well and results were even better than anticipated.
Collect results from the pilot and validate the feasibility of the framework based on them. Adapt requirements based on the new information.	Pilot experiences were collected together in Chapter 5 and based on them the overall framework was declared feasible. Some changes to the framework were also suggested, the biggest one being the decision to use only the keyword-driven approach. Changes affected also detailed requirements specified earlier and a revised requirement set was presented.

Table 6.1: Evaluating the thesis against its requirements

Bibliography

J. Bach. Agile test automation, 2003. URL `http://www.satisfice.com/agileauto-paper.pdf`. April 11, 2005.

A. Bagnasco, M. Chirico, A. M. Scapolla, and E. Amodei. XML data representation for testing automation. In *IEEE AUTOTESTCON Proceedings*, pages 577–584. IEEE Computer Society, 2002.

K. Beck. Simple Smalltalk testing: With patterns, 1994. URL `http://www.xprogramming.com/testfram.htm`. April 13, 2005.

K. Beck. *Test-Driven Development By Example.* Addison-Wesley, 2003.

K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, July 1998.

C. Bird and A. Sermon. An XML-based approach to automated software testing. *ACM SIGSOFT Software Engineering Notes*, 26(2):64–65, March 2001.

BS 7925-1. *Glossary of terms used in software testing.* British Computer Society, 1998.

I. Burnstein. *Practical Software Testing.* Springer, 2003.

H. Buwalda, D. Janssen, and I. Pinkster. *Integrated Test Design and Automation: Using the Testframe Method.* Addison-Wesley, 2002.

R. Craig and S. Jaskiel. *Systematic Software Testing.* Artech House Publishers, 2002.

S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 285–294. IEEE Computer Society Press, 1999.

E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing.* Addison-Wesley, 1999.

M. Fewster and D. Graham. *Software Test Automation.* Addison-Wesley, 1999.

M. Fowler. Separating user interface code. *IEEE Software*, 18(2):96–97, March 2001.

A. Geras, M. Smith, and J. Miller. A prototype empirical evaluation of test driven development. In *METRICS '04: Proceedings of the Software Metrics, 10th International Symposium on (METRICS'04)*, pages 405–416. IEEE Computer Society, 2004.

P. Hamill. *Unit Test Frameworks.* O'Reilly, 2004.

A. R. Hevner, S. T. March, J. Park, and S. Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, March 2004.

IEEE Std 610.12-1990. *IEEE standard glossary of software engineering terminology.* Institute of Electrical and Electronics Engineers, Inc., 1990.

R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

C. Kaner. Pitfalls and strategies in automated testing. *IEEE Computer*, 30(4): 114–116, April 1997.

C. Kaner, J. Bach, and B. Pettichord. *Lessons Learned in Software Testing: A Context-Driven Approach.* John Wiley & Sons, Inc., 2001.

M. Kelly. Choosing a test automation framework, July 2003. URL `http://www-106.ibm.com/developerworks/rational/library/591.html`. April 30, 2005.

E. Kit. Integrated, effective test design and automation. *Software Development*, pages 27–41, February 1999.

M. Last, M. Friedman, and A. Kandel. The data mining approach to automated software testing. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 388–396. ACM Press, 2003.

B. Marick. Bypassing the GUI. *Software Testing & Quality Engineering*, 4(5):41–47, September 2002.

B. Marick. Classic testing mistakes, 1997. URL `http://www.testing.com/writings/classic/mistakes.html`. April 30, 2005.

E. M. Maximilien and L. Williams. Assessing test-driven development at IBM. In *Proceedings of the 25th International Conference on Software Engineering*, pages 564–569. IEEE Computer Society, 2003.

A. M. Memon, M. E. Pollack, and M. L. Soffa. Using a goal-driven approach to generate test cases for GUIs. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 257–266. IEEE Computer Society Press, 1999.

A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 30–39. ACM Press, 2000.

G. Meszaros. Agile regression testing using record & playback. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 353–360. ACM Press, 2003.

D. Mosley and B. Posey. *Just Enough Software Test Automation.* Prentice Hall PTR, 2002.

R. Mugridge and W. Cunningham. *Fit for Developing Software.* Prentice Hall PTR, 2005.

C. Nagle. Test automation frameworks, 2000. URL `http://safsdev.sourceforge.net/DataDrivenTestAutomationFrameworks.htm`. February 20, 2005.

J. Ousterhout. Scripting: higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

B. Pettichord. Deconstructing GUI test automation. *Software Testing & Quality Engineering*, 5(1):28–32, January 2003.

B. Pettichord. Seven stepts to test automation success. In *Proceedings of the Software Testing, Analysis & Review Conference (STAR) West 1999.* Software Quality Engineering, 1999.

B. Pettichord. Design for testability. In *Proceedings of The 20th Annual Pacific Northwest Software Quality Conference*, pages 243–270. Pacific Northwest Software Quality Conference, 2002.

B. Pettichord. Hey vendors, give us real scripting languages. *StickyMinds*, February 2001.

R. W. Rice. Surviving the top ten challenges of software test automation. In *Proceedings of the Software Testing, Analysis & Review Conference (STAR) East 2003*. Software Quality Engineering, 2003.

D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 105–118. ACM Press, 1992.

H. Robinson. Intelligent test automation. *Software Testing & Quality Engineering*, 2(5):24–32, September 2000.

M. Shaw. What makes good research in software engineering? *International Journal on Software Tools for Technology Transfer*, 4(1):1–7, October 2002.

R. Strang. Data driven testing for client/server applications. In *Proceedings of the Fifth International Conference of Software Testing, Analysis & Review*, pages 389–400. Software Quality Engineering, 1996.

L. H. Tahat, B. Vaysburg, B. Korel, and A. J. Bader. Requirement-based automated black-box test generation. In *Proceedings of the 25th Annual International Computer Software and Applications Conference, 2001*, pages 489–495. IEEE Computer Society Press, 2001.

K. Zallar. Are you ready for the test automation game? *Software Testing & Quality Engineering*, 3(6):22–26, November 2001.

K. Zambelich. Totally data-driven automated testing, 1998. URL `http://www.sqa-test.com/w_paper1.html`. February 20, 2005.