# Testing Advanced Web Interfaces

*Master's Thesis*

Justin Boekestijn

# Testing Advanced Web Interfaces

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Justin Boekestijn
born in Delft, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

TOPdesk
Martinus Nijhofflaan 2
Delft, the Netherlands
www.topdesk.com

# Testing Advanced Web Interfaces

Author: Justin Boekestijn
Student id: 1047205
Email: j.boekestijn@student.tudelft.nl

**Abstract**

AJAX-based Web applications have gained massive popularity in the last few years, but tool-assisted testing of their user interfaces is an area lagging behind thus far. While these interfaces show similarities with other UI technologies, unique aspects of this architecture necessitate research into approaches suitable to create an automated testing framework. In this report, an attempt is made at implementation of a tool prototype, designed in support of a specific AJAX-based interface toolkit; this tool is based upon the concept of controls as implemented in this type of toolkit. Its prototype is used to test a part of an application built upon this toolkit, then generalized to be able to support other toolkits.

Thesis Committee:

| | |
|---|---|
| Chair: | Prof. Dr. Arie van Deursen, Faculty EEMCS, TU Delft |
| University supervisor: | Ir. Ali Mesbah, Faculty EEMCS, TU Delft |
| Company supervisor: | Ing. Roel Spilker, TOPdesk |

# Preface

I would like to thank several people for their cooperation in this project. First of all, my company supervisor Roel Spilker and university supervisor Ali Mesbah, for guiding me through the project. Then, the people I interviewed at TOPdesk while gathering requirements: testing team members Mark Rotteveel and Marc de Hoop, developer Vincent Zorge and Mango developer Robert Amesz. Finally, I want to thank everyone who took my tool for a test drive: TOPdesk's testing team members, and especially my personal software tester, Manon Boekestijn.

Justin Boekestijn
De Lier, the Netherlands
December 16, 2008

# Contents

# List of Figures

# Chapter 1

# Introduction

The Web has been evolving ever since its inception, with one of its more recent achievements the set of technologies and techniques dubbed AJAX, Asynchronous JavaScript And XML [16]. For the last few years, AJAX has transformed the Web platform from an interconnected information medium to a part of a serious application platform. These applications, in most cases using little more than a Web browser as their client platform, are becoming a serious competition to classic applications built on a specific OS or virtual machine architecture. AJAX technologies have aided this process by fronting these applications with interfaces equaling or surpassing the richness of classic applications. Complex as they are getting, serious user interface testing needs to be done to ensure their quality; to improve efficiency, this process can be supported by automated testing tools. As of yet, tools to accomplish this goal are not suited for use with this relatively new type of graphical user interface. Particularly, current tools are focused on testing semi-static HTML Web sites rather than complex Web application interfaces relying on single-page interaction instead of URL-based interaction.

## 1.1 Problem statement

Testing tools for classic applications[1] have been around for years, supporting all sorts of programming languages and development interfaces. Although tools also exist for Web applications with a technically more limited user interface, these are generally focused on testing whether certain input sequences to forms and hyperlink clicks generate errors on the server side of the application. As for our object of interest — rich, JavaScript-ridden interfaces — no widely adopted options seem to exist if one wants to automate testing in a robust manner. Changing this is by no means a trivial task; but current research looks promising, and because Web interface development is becoming a more structured activity when using tools such as interface builders with an extensive set of interface components, it seems very well possible to build an interface testing tool capable of handling structured AJAX-based applications.

[1] I will refer to applications built for a specific operating system or virtual machine as a ”classic application”

## 1.2 Motivation and goals

One interface toolkit in particular has sparked interest in researching this problem: Mango, currently in development by Delft-based company TOPdesk for their eponymous service management application. Over the last few years, TOPdesk has been transformed into a pure Web application, incorporating AJAX-inspired features in its interface. This transformation has culminated in the search for a fitting interface toolkit to further aid development of TOPdesk. After evaluating several existing toolkits, some of which are discussed in this report, TOPdesk's development team decided to develop their own kit, which they have dubbed Mango. At the time of writing, Mango has entered a phase in which some sections of TOPdesk's interface have been reimplemented as Mango pages, with more conversions planned for each future release. Their desire is to have a functional regression testing tool able to interact with any Mango-based application, to be used by developers building these applications. Points of focus and requirements set by TOPdesk are treated in chapter 3.

### 1.2.1 Scope

I will try to keep this research as general as possible regarding the range of interfaces which may be tested with an AJAX testing tool. Most of it will assume a well-defined interface toolkit has been used to develop the application under test, to be able to keep an analogy with classic applications. However, like every classic interface toolkit is different, AJAX toolkits range widely in their features and approaches, which will result in some concessions regarding generalization — I will discuss this in more detail in chapter 7.

## 1.3 Research questions

A variety of aspects needs to be researched before being able to set up design and implementation of a generalized AJAX testing tool. First of all, a demarcation needs to be made regarding the definition of interface testing. Will functional testing be the only thing considered, or might performance and perhaps usability testing be incorporated in the same tool? Then, investigation must be done in testing philosophies and methods. Which approaches have been used within the world of software testing? Much may be derived from classic interface testing and ”regular” Web interface testing, also with regard to technical approaches: how will this tool fit into the development and build environment, how will it interact with the application? And although current tools may be inadequate for AJAX needs, some might be promising enough to build the desired tool upon. This leads to a next question, important in selection of a suitable tool to extend: which features need to be included in the tool? Last but not least, how will a testing tool designed for use with Mango be able to cope with other interface toolkits? All in all, a lot of aspects needed to be researched, leading to this report.

## 1.4 About TOPdesk development

While TOPdesk's last incarnation, still in use by many of the company's clients, was built in FoxPro exclusively for the Windows platform, the application's latest major version has been designed as a pure Web application. This gradual transformation started by developing a Web interface for the application's most widely used features in support of the FoxPro client. This Web interface was built upon the Jetty Web server[2] using Java servlets and JSPs. After many features of the Windows application had been incorporated into the Web interface, work started on a TOPdesk version which could operate completely independent from FoxPro and Windows. This Web application has been built upon the same Java technologies as its predecessor, but supports storage in several popular database types like Microsoft SQL Server and Oracle. Its user interface has also been completely redesigned, adding many AJAX-related features to improve the perception of working with a regular application.

### 1.4.1 Current state of testing at TOPdesk

At the moment, a quality assurance team of about fifteen employees ensures the number of software bugs in TOPdesk is reduced to a minimum. Apart from these human testing efforts, unit tests are performed regularly to perform regression testing, and nightly builds are generated from the source code repository to make sure everything compiles properly. Although the QA team is a valued asset of the company, the development team is constantly researching options to improve and expand automated testing procedures. Interface testing is one of some less explored aspects of these procedures.

[2] `http://jetty.mortbay.org/`

# Chapter 2

# Related Work

## 2.1 Testing aspects

Testing encompasses a wide variety of procedures and activities to ensure and improve software quality. The majority of related research with respect to graphical user interfaces has focused on three particular aspects: functional, performance, and usability testing. Functional testing, also known as black-box testing or specification-based testing [19], is the most fundamental part of testing, that verifies whether an application works as specified by its design. Performance testing is used to measure whether these application features execute within a reasonable amount of time; functional test cases may be reused in this procedure, although these do not generally cover every detail of performance testing [23]. Both of these aspects are a subject of increasing automation, and are interesting to automate in an AJAX interface context. The third aspect, usability testing, is largely dependent on human interface guidelines and user input; automation of certain elements of usability testing is possible, such as evaluation of experiment results [10]. Most other elements require input from users other than developers, as their approaches to usage of an application differ in most cases [24], making them a valuable source of usability assessment data.

### 2.1.1 Focal points

Because of the crucial importance of functional testing, it will be the main point of interest in this research. Performance testing may be implemented as a future extension, when Mango has reached a maturity level at which the time is ready to tweak its performance. Usability testing is a completely different aspect, and will be left out for the remainder of this report.

## 2.2 Procedural approaches

Functional testing may be executed using several different approaches, most of which are also applicable to AJAX interfaces. One widely known approach, also used in testing of HTML pages [18], is static analysis. Although this is a useful method for testing static interfaces, the dynamic nature of AJAX and the variety of programming languages used

to implement this type of interface make static analysis unsuitable for testing AJAX-based interfaces. Other methods can be roughly divided into two categories, referring to their approach to achieving test coverage: methods related to record-and-playback expand coverage by creating tests on a case-by-case basis manually, while methods based upon finite state machines start by automatically exploring every possible path through an application, and manually filtering out cases considered irrelevant.

Record-and-playback, alternatively known as capture-and-replay, is a popular and intuitive approach to functional UI testing, which requires a programmer or QA employee to define test cases by interacting with the application's user interface, while being monitored by a tool recording the corresponding test scripts [7]. This technique suffers from several issues: the degree of possible automation is limited due to the requirement of user interaction while recording scripts; manual effort is also needed in tweaking recorded scripts to match intended interaction, and in adding verification steps to check whether actual results match expected results. Also, because these scripts are created manually, coverage may be limited to the imagination of the script writers. These issues may be partially solved by capturing actual user sessions [20] to generate test scripts from; manual effort will then primarily consist of filtering useful test cases from these sessions.

On the other hand, methods based on finite state machines focus more on automation of achieving maximum coverage by using a tool to extract every possible interface state from the application [1] [2]. Several techniques for state extraction have been defined in related research, based on concepts such as plan generation [13], GUI ripping [12], and UML diagrams [22]. Manual effort is still needed in limiting the input space to relevant states, as well as defining verification steps; however, plugin interfaces are available for some FSM-based tools, allowing reuse of generalized verification actions which are not specific to the application under test.

The oracle verification issue has also been the topic of a great amount of research, digging into requirements for consistent and complete validation of an application, along with methods to automate insertion of verification steps at appropriate moments. Issues encountered during this research are related to, among other aspects, keeping test results consistent while changing case execution order [6], changing execution context [9], concurrent script execution, and nondeterministic application features [21].

## 2.3 Testing of other GUI technologies

Existing research on automation of user interface testing in applications focuses mainly on classic applications — those built for a specific operating system or virtual machine — while research on Web interface testing is aimed largely at Web sites, be they static or dynamic, other than those used to front applications. Some parallels can be drawn between testing classic application interfaces and Web application interfaces with respect to methods used; the differences between the building blocks used to assemble these interfaces are the most important obstacle in comparison of these technologies.

The original set of Web programming and markup languages have never been intended to be used to replace traditional applications [17]; attempts to develop new Web standards

for creating Web application user interfaces [4] have failed, possibly due to the popularity of current standards and their extensions. Web developers have therefore been forced to build one or more layers upon HTML and JavaScript in order to facilitate application development; however, these layers are not yet used by any testing tool [3], instead referring directly to DOM elements as components to test. A challenge lies in creating a tool which is able to abstract away from the DOM, referring to virtual controls introduced by the UI toolkit layer used to implement the application interface; this is particularly difficult, because most Web UI toolkits generate client code not directly retraceable to this control layer. However, this type of tool is desireable because the control concepts used in generated test cases correspond to both concepts used in interface development and visual elements, as presented to the user. The availability of a hybrid tool, capable of handling both controls and custom HTML used in the interface, is also desireable, as most Web interface toolkits allow developers to insert arbitrary HTML between sections of controls; this code also needs to be tested thoroughly.

### 2.3.1 Extending an existing tool

Web interfaces have evolved in a way where existing technologies are used while introducing new techniques rather than employing new browser features. Most existing testing tools, however, are designed with Web 1.0 in mind, supporting only a limited set of scripted interface automation. Nevertheless, it should be possible to extend an existing Web interface testing tool to support more sophisticated scripted interfaces. And while Web 2.0, with all its AJAX beauty, appears to be a set of completely new technologies, it is still based on Web 1.0. Thus, features available in Web 1.0 testing tools are still very useful to build upon, although it should be kept in mind that the evolution of the Web from an information platform to an application platform [8] has also shifted the focus on testing needs.

### 2.3.2 Current Web testing tools

The open nature of Web technologies allow for a wide variety of methods to use these technologies in application interface development. Testing methods used for Web interfaces also show a broad variation, of which three options tend to be the most popular among test tool developers. One of these methods uses a HTTP proxy to monitor or manipulate interaction between the client interface at the browser and the application code executing at the server side. This method, seen in tools such as TestMaker[1] and Sahi[2], is limited in its inspection and interaction capabilities because it is in direct contact with neither server nor client, and is useful mostly for testing static Web sites. Another widely implemented approach uses a browser emulator to maximize aforementioned capabilities; this method has been used in Canoo WebTest[3], based on the HtmlUnit emulator. Although emulators have full control over the client interface, this approach is unsuitable for testing complex AJAX applications, as the emulator will fail to detect application errors caused by quirks found

[1] `http://www.pushtotest.com/`
[2] `http://sahi.co.in/`
[3] `http://webtest.canoo.com/`

in popular browser implementations. Testing methods relying on manipulation of actual browser instances are therefore much more interesting to employ in an AJAX interface context; implementors of this approach either embed an automated instance of a browser in their tool, such as Badboy[4] and Watir[5], or embed their tool into the browser as a plugin, such as Selenium[6], specifically its IDE component. A disadvantage of these methods is the fact that different implementations are needed to support different browsers; however, this issue is favored over the interaction limitation found in proxies and the additional browser introduced by an emulator.

All of the tools mentioned above operate by executing test scripts defined by a user, providing either a custom scripting language or an API to their test execution engine for test case definition. In some cases, a test recorder is included to complete the capture-and-replay paradigm. However, current research also examines methods requiring less input from users in test creation. For instance, a tool named Atusa [15] has been developed, which is able to verify correctness of an AJAX user interface by feeding interface states to a plugin mechanism. These interface states are determined by crawling the application with a specialized tool, Crawljax [14]. Plugins are defined manually, and are meant to verify invariants instead of application-specific features.

## 2.4 Web interface toolkits

As with classic applications, a variety of toolkits is available to implement AJAX GUIs. Each has its own approach with regard to, among other aspects, division of code execution at the client and the server, division of responsibility and integration with back-end code. They have one thing in common: all of them have been developed for, to quote Google, ”developers who don’t speak browser quirks as a second language”[7].

One of the research-wise most challenging issues in developing an AJAX testing tool, but also one of the most desirable features, is the ability to deploy this tool for as many different toolkits as possible. As these kits are largely heterogeneous, some concessions will have to be made to keep the project feasible. A pluggable framework will probably be the most intuitive solution for tool implementation; toolkit components should be defined on a plugin interface, specifiying their methods of interaction with the outside world. After tool implementation for Mango, generalization of the testing tool will be based on several toolkits currently in general use; they are either interesting because of their reputation or because a testing tool is already available for them. In particular, Google Web Toolkit[8] and Echo[9] will be used for implementation. Details on these toolkits and this phase are explained in chapter 7; for now, an explanation of a toolkit already supported by a customized testing tool is in order.

[4] http://www.badboy.com.au/
[5] http://wtr.rubyforge.org/
[6] http://www.openqa.org/selenium/
[7] http://code.google.com/webtoolkit/
[8] http://gwt.google.com/
[9] http://www.nextapp.com/platform/echo2/echo/

### 2.4.1 TIBCO

TIBCO[10] features a fully-fledged IDE, built using its own toolkit, to enable developers to define their GUI by implementing JavaScript and dragging together interface elements. Generated code includes wrapper elements specifying user-defined component labels along with generated ids and classes; these identification attributes are used by GITAK[11], TIBCO's testing tool based on Selenium, to recognize and reference components used in test scripts.

This method of locating elements sketches the ideal situation for a GUI testing tool: identification by a label chosen by a developer is unambiguous and less error-prone than using generated id's. Readability of test scripts is also improved, as are reusability and maintainability.

[10] `http://www.tibco.com/software/rich_internet_application/general_interface/`
[11] `http://www.tibco.com/devnet/gi/product_resources_gitak1.jsp`

# Chapter 3

# Requirements

## 3.1 Decisions based on related work

An important requirements-related decision that needs to be taken with regard to tool implementation is one for which options have already been mentioned in section 2.3. Research of literature and existing tools has shown an unexplored area in AJAX testing with this regard: a shift of focus in creation of a testing tool from directly exposing the HTML layer of an application to the user to usage of a more abstract layer of virtual controls, as they appear in Mango and other toolkits. Using this concept will result in a tool comparable to regular application UI testing tools, which also focus on interface components, rather than their underlying implementation. To explore this path is therefore the focus of further research in this report.

## 3.2 Involved parties

The most important stakeholders involved in successful development of a testing tool are obvious: the entire development department shares an interest in more efficient regression testing. When the interface toolkit is being developed in-house, these developers may be divided into two categories: those developing the toolkit itself, and those developing code using this toolkit. However, other parties may also benefit from the achieved automation, parties more directly involved with clients; at TOPdesk, these include the bespoke work department and the help desk. Interviews conducted with selected members of these teams have resulted in a list of requirements used in selecting a testing tool to extend, and in implementation of the tool extension.

### 3.2.1 Development

For a developer, it is necessary to be able to know as soon as possible whether a change or addition made to program code will break existing functionality; at the moment, most of these issues are discovered during test cycles performed by the quality assurance team. Being able to quickly run some regression tests after any significant code modification would enable developers to fix these problems sooner: after all, a bug discovered sooner is more

easily retraceable to a recent change in code. At the moment, TOPdesk developers perform interface testing in a purely manual way, followed by a request to the software testing team to do some more manual testing. Automation has been carried out in back end testing by means of a collection of JUnit test cases, maintained by developers, which are executed daily.

Because Mango is developed in-house, developers share a responsibility with regard to ensuring application tests can be set up in a convenient way. Keeping in mind the fact that the testing tool should be extendible to suit other interface toolkits, for which source code or even modifications in support of this convenience are not available, it seems rational to keep Mango toolkit developers out of the loop in development of the testing tool. However, seeing as both developers and end users would benefit from code more suitable for automated testing, it is more practical to also have Mango toolkit developers influence requirements used for the testing tool.

#### Mango development

A view expressed by one of Mango's developers is significantly different from other opinions on the necessity of a Mango testing tool. Its deviation is mainly based on the assumption that it is not necessary to test the entire Web application from user interface to business logic and data integrity with a single tool and set of test scripts. Instead one could test the application starting at the layer directly below the user interface through a Web services interface, and test only the components used in generated HTML-based interfaces separately. Because these components are tested one at a time, an abstraction layer would not be needed.

However, the downside to this approach is the lack of testing done on the actual application interface: the aggregate of a set of user interface controls backed by business logic. While components themselves may be bug-free, their use within a user interface introduces different bugs, among which are those caused by components interfering in ways overlooked while testing the user interface toolkit itself. For this, the component abstraction layer forms a welcome addition.

On a side note, an assurance from the Mango development team has been made that Mango is easily extendible to provide more component information to the testing tool. This would in practice mean traversal of several layers of Mango's implementation to retrieve desired information.

### 3.2.2 Quality assurance

The majority of TOPdesk's testing activities are performed manually by a team mainly made up of students. Automation of these tasks has been attempted on several aspects, but these efforts have concentrated on back end testing. A significant part of a software tester's time at TOPdesk is spent doing regression testing by clicking through the application interface. This valuable time can be saved by having a tool perform these tasks on a daily basis. Recently, one of the team members has started development of an interface testing tool based on image recognition in his spare time. Although this approach has the advantage of

independence from application implementation details, this method of abstraction yields a tool also lacking the fine-grained control needed to interact with and inspect an advanced application interface.

One of the most important criteria for a testing tool used by TOPdesk's testing team is its ease of use: the average team member has only a basic understanding of programming, which means creation and maintenance of test scripts should be kept as simple as possible. The current implementation of the image recognition tool mentioned above does not adhere to this requirement, instead providing a programming interface for Java. A more suitable solution would be the inclusion of a basic scripting language, supplemented by a test recorder which is able to generate test scripts in this language. Another option would be to leave interface test automation to developers, as is already the case with the set of JUnit tests currently in use. Tool selection will also depend on available feedback features. The testing team has expressed a need for detailed feedback in case of test failure, also when these tests are run unsupervised.

These software testers are the main reason for selecting a particular automated testing paradigm: due to their unfamiliarity with complex software engineering concepts, a rational choice would be to go with the most intuitive approach. This approach, record-and-playback, is unsurprisingly also the most popular method of automated user interface testing, when counted by tools available. Although this means recording and maintaining test scripts still involves a certain amount of manual effort, it is undoubtedly an improvement compared to purely testing a Mango interface by hand. This manual effort stems in part from the need to define scenario-specific verifications for each script created; because this is an aspect difficult to automate — which is discussed in further detail in section 6.2.3 — the advantage of creating scripts by means of state graph generation is lost when adding these application-specific verifications. Also, this project aims to reduce maintenance effort by using a script syntax designed to be more robust combined with AJAX toolkits such as Mango; without this addition, testing Mango interfaces aided by tools would be impossible due to its generated HTML code, an issue that will be further highlighted in section 6.2.2.

### 3.2.3 Bespoke work

Many of TOPdesk's clients have requested some addition or customization to their installation, which in turn has been implemented by TOPdesk's bespoke work department. A major issue with regard to maintenance of these customizations is the fact that they may break, whenever a new version of the software is installed at the client. Ever since this bespoke work was introduced, a more proactive approach has been desired to prevent such breakage. However, because of the nature of current bespoke work implementation being very irregular, it is infeasible to have each customization tested automatically.

The introduction of Mango will hopefully streamline implementation methods, and create a more structured set of client-requested customizations. As a result, it should be easier to execute a test suite verifying that each piece of bespoke work interface still functions as expected after a change to TOPdesk code. If something breaks, a solution may be found in a fix to this code, before a new version is released to the client.

Furthermore, the bespoke work team will also benefit from extendibility of the tool

itself to interact with customized Mango controls. Although bespoke work implementation on Mango has not yet begun, past experience has shown that custom controls are often needed to suit client demands. This issue is also related to extension of the tool to other interface toolkits.

### 3.2.4 Help desk

As a support department to clients, bug reports are frequently submitted to the help desk. It is their job to reproduce these bugs and to notify developers whenever a bug is reproducible. A testing tool may facilitate this process by allowing the help desk employee to record a reproduction script, in order to send this to developers. This saves time at the help desk used to describe these reproduction steps verbosely in a bug report. Because this type of use is only of secondary importance, and because of the fact that the number of people involved in requirements gathering should be limited, no requirements inventory has been done with help desk employees.

## 3.3 Use cases

The component-based interface testing tool will most likely be implemented as an extension to an existing tool. This more or less restricts use cases to a listing existing uses of these tools, because the aim of this project is not to significantly alter interface testing procedures. Instead, the abstraction layer introduced is meant to create more robust and comprehensible test scripts; use cases listed here are mostly deducted from existing tool features. In all relevant cases, the assumption is made that the server hosting the application under test is already running.

### 3.3.1 Creation of a new test script

The use case is initiated by a test suite maintainer starting a Web browser and its test script recording extension. By starting the recorder, any page navigation and in-page action done by the user will be recorded into a new test script. The user then navigates to the URL used to access the application under test and interacts with the application. When all desired interactions have been executed, the user orders the script recorder to stop recording by clicking the appropriate button on its interface. At this point, the user inserts assertions to check the user interface state after some of the recorded actions. To verify its correctness, the user has the recording extension run the resulting test script. If the user's expectations with regard to test execution are satisfied, he saves the script to disk. Otherwise, the process of modifying an existing test script, as described below, may be followed until results are satisfactory.

### 3.3.2 Modification of an existing test script

This use case is similar to creation of a new test script; actors are identical and actions involved are comparable. The scenario again starts with a test suite maintainer opening

a Web browser and test recorder window. The user then opens a previously saved test script. He instructs the recorder to stop test execution at a certain instruction by placing a breakpoint at the appropriate location. The user then orders the recorder to run the test script; it will halt execution when the breakpoint has been reached. At this point, the user starts recording actions by clicking the record button, then interacting with the application under test; the script recorder inserts new actions at the breakpoint location. When the user has finished these interactions, he clicks the record button once again to stop recording. The user may also insert new assertions between any two script actions. Also, script actions themselves may be modified manually. Once modifications are done, the user saves the script to disk.

### 3.3.3 Writing a test script in a higher order language

A script maintainer initiates this use case from one of two starting points: either a script will be written from scratch, or an existing test script, written in the regular test script definition language, will be extended. The first option is essentially a pure programming task, which needs no further detail in this use case. The second option is aided by a conversion tool. The user loads an existing script into the conversion tool, after which he is presented with a select set of programming languages to which the script may be converted. After choosing one of these languages, the loaded script is converted into source code suitable for the selected language; this code uses an API provided by the testing tool, which is able to control the test runner engine. The generated code may be altered at will, but this is again a pure programming task.

### 3.3.4 Manual execution of a test suite

The test suite maintainer is the actor in this case; this person may or may not be a script maintainer. The use case is initiated by starting the test runner application. After the user loads the test suite into the test runner, the test suite is ready to be executed. The user presses the run button to start the test suite; the test runner reports failure and success messages for each test script executed.

### 3.3.5 Scheduled execution of a test suite

This use case involves a test suite maintainer and either an external scheduling or a continuous integration service. The most basic approach consists of the user instructing the scheduling service, such as cron or Windows' Scheduled Tasks, to run a certain command at a specified time. Execution of this command launches the test runner, which then runs the configured test suite unsupervised for each browser platform supported and requested by the user. Reports on test failure and success are stored or forwarded by the test runner for later inspection.

A more advanced approach adds test execution to the continuous integration cycle of the software. This alleviates the tasks of scheduling test execution and processing separately provided test results from the user; the use case changes into a one-off activity of performing the integration of the testing tool with the continuous integration framework.

## 3.4 Breakdown by importance

### 3.4.1 Must have

When focusing on the average test team employee as an end user — one of two user groups deemed most relevant, alongside developers — an absolute necessity with regard to tool features is its ease of use. To aid these users in creation and maintenance of test cases, the tool must have an intuitive script recording interface which uses a test case definition language comprehensible by non-developers. This interface and language must also provide extensive capabilities for specifying assertions, needed by end users to verify test execution correctness. Where applicable, error messages generated by these assertions must adhere to the terms and metaphors introduced by the component abstraction layer.

Furthermore, in order to be able to integrate automated interface tests into a company's nightly building and testing process, support is needed for completely unsupervised test execution. To achieve this, two criteria need to be met: one must be able to schedule test execution running in a form that does not require user intervention, and one must have the opportunity to read detailed test results after execution at any time, preferrably also at any place. One particularly elegant and versatile approach to satisfy both requirements is to integrate test execution and feedback with the continuous integration cycle in use for the application under test. Another approach to satisfy the second criterium is allowing the user to specify an URL for the testing tool to call whenever a test result is available, to which the tool appends some parameters; this URL should point to a script processing the provided test results in a way specified by the user.

A requirement which is also valid for user interface toolkits themselves is the extendibility of the tool to support custom interface controls. This requirement is related to the problem of extending the tool to facilitate support for completely different Web user interface toolkits; the layer placed on top of the layer responsible for HTML and JavaScript interaction must be sufficiently extendible, so that interface toolkit developers are able to define the interaction models needed for their components.

### 3.4.2 Should have

While developers are also able to use a simple test case definition language, their demands with regard to test case complexity may soon exceed what is possible without a fully-featured programming language. Therefore, there should be a way for developers to describe more elaborate test cases. Ideally, this would be implemented by providing an API to the testing engine in an existing language. In practice, most tools provide either an API or a definition language, making this requirement conflict with the ”must have” of a tool that is easy to use.

### 3.4.3 Could have

The first feature that would be nice to have is an extension to both the script recorder and the extendibility of the tool itself. To further ease the process of adding recognition of new interface components to the tool, a wizard could be developed to generate component

definitions for use by the testing tool. This wizard would ask the user to point out the custom component in a live application environment, followed by a possibility for the user to execute some actions upon the component and name these actions according to suitable terminology. For example, if a user wants to add a definition for a date selection component, he would first be asked to identify component elements by clicking them, then be allowed to specify actions such as `clickNextMonth` by executing and naming them. The feasibility of the implementation of this wizard is largely dependent on implementation details for the component layer itself.

Streamlining user interface testing with other automated testing activities would also be a welcome addition to the feature set. Some tools implement this integration by providing a task for frameworks such as Ant or NAnt to execute test cases. Others, in particular those tools designed to be programmed directly from an existing language, are easily integrated into the unit testing framework already used for other testing purposes.

### 3.4.4 Won't have

Features the tool will not have primarily belong in the category of features not usually associated with the capture-and-replay paradigm. These include the traversal of state graphs and involvement of GUI models in test case generation.

# Chapter 4

# Tool selection

## 4.1 Derived requirements

### 4.1.1 Extendibility

If a tool is to be extended at all, the extension options available within the original tool should be the primary criterium in tool selection. A significant amount of modification will be needed in order to achieve the goal of a component-based Web interface testing tool. This implies that a completely modifiable open-source solution is preferred over a proprietary tool which provides a limited API.

Also, TOPdesk has expressed a desire to release the testing tool as an open source project. This means that a solution already supported by the open source community is preferred, in order to increase the odds of the extension being picked up by the community.

### 4.1.2 Browser support

The aspect of browser support can be satisfied in at least two ways, either by writing a separate tool for each supported browser or by building a single tool which includes code to take care of special cases applicable to a specific browser. TOPdesk guarantees compatibility with a limited number of browsers: specifically, Microsoft Internet Explorer 6 and up and Mozilla Firefox 1.5 and up are supported. At the time of writing, both of these browser versions have already been superseded by version 7 and 3.0, respectively; support for these older browser versions in a testing tool is therefore not a requirement per se, but rather a feature that would be nice to have.

The first implementation option, supporting each browser with a separate tool, is not just a tedious task for the tool developer, but also for the test maintainer. Therefore, finding a tool that already supports multiple browsers is preferrable. This criterium eliminates many tools only suitable for testing applications with Internet Explorer, the browser with the largest market share at time of writing. A suitable framework would support not just Firefox and Internet Explorer, but also browsers that might be added to TOPdesk's compatibility list in the future, such as Apple's Safari.

### 4.1.3 Ease of use

Like with every piece of software, a balance needs to be found between tool versatility and simplicity. This balance is largely dependent on the target audience, which in this case is a diverse group. Testing personnel is not necessarily proficient at programming, which means scripting of test cases should be as simple as possible. On the other hand, developers usually have enough knowledge of programming to be able to implement test cases in a fully-fledged programming language. The majority of current testing tools is limited to use either a scripting language developed specifically for the tool itself, or to provide an API for one or more existing languages. However, there are some exceptions to this rule: these tools generally provide a very simple test case definition language, that includes commands for interaction with the application, along with assertions for inspection of the application state. More advanced test programming support is then offered through a library disclosing these commands.

### 4.1.4 Automation

When selecting capture-and-replay as the automated testing approach, one will need to put considerable effort into creation of test cases. This project aims to reduce this effort by improving test script syntax to be more robust, and improving test script recorders to capture intended user interaction. As a requirement shared with the extendibility argument, the tool needs to be selected on extendibility of both syntax and recorder. Another aspect of script recording which allows for (at least partial) automation is the inclusion of assertions and verifications to recorded scripts; while this phase of script writing is still a matter of selecting relevant checks to add, the tool should be able to at least offer suggestions based on recorded actions and monitored page content changes.

Apart from enhancing test creation, the tool also needs to support automation of test execution; more specifically, tests need to be executed unsupervised, and test results need to be available afterwards. Unsupervised execution may be achieved in several ways: one option is to keep a service process running continuously, executing the test suite whenever a signal is received through a predefined protocol. Another, simpler option is to start the tool through a special command, which makes it run its test suite automatically; this command can be added to a scheduling service to execute tests on a periodical basis. Implementations of this second option can be found in nearly every testing tool, which makes this requirement less of a worry when eliminating candidates for tool extension.

Communicating test results for later use can be done in a multitude of ways. The most basic way of saving test results is of course done by providing a log file containing these results after execution, which may then be analyzed manually or by external tools. A tool may also provide an option to automatically send these test results through email or, generalizing this concept, through a script written by the user, providing a much more flexible solution for result processing. Another option is the inclusion of special reporting actions within scripts, providing a means to store test results selectively. All of these issues are, to an extent, only applicable to stand-alone tools; those integrated into other testing frameworks and those consisting purely of a browser automation API can shift responsibilities

regarding both execution and feedback to the encapsulating framework, such as JUnit or NUnit.

A related point of attention is the level of detail incorporated into test results. Saving detailed failure information is a valuable aspect of automated testing, especially when application states resulting in test failure are not easily retrievable. Although showing the number or a list of passed and failed tests is a start, useful test results also show information regarding application state and error messages at time of failure. In case of a Web application, the state information should contain at least a dump of the DOM tree; values of JavaScript variables and a screen capture of the application interface may also aid investigation into test failure. Error messages should be propagated from corresponding messages produced by the application itself, where possible, supplemented by messages generated by the testing tool; these messages should indicate point of failure and a reason for failure.

### 4.1.5 Testing approach

The ease of use criterium also partially determines the choice of tool by filtering candidates based on their testing approach: although both capture-and-replay and FSM tools require manual effort to create and tweak sensible test cases, FSM tools tend to require input from users with a large amount of expertise in the field of computer science. The tool requested by TOPdesk is to be used by their current software testing personnel; a capture-and-replay tool will suit their level of expertise better than an FSM tool.

Otherwise, the approach chosen is of minor importance, as goals set for this project are relevant for both methods. Improving robustness of test case definition is an issue relevant whenever a highly dynamic application is involved, while automation of the introduction of verification steps is an idea applicable to any testing tool.

### 4.1.6 Legal issues

In order for the extended tool to be deployed in a commercial environment, licensing issues need to be taken into account. These issues are twofold: first, a license for use of the original tool is needed; a tradeoff between license cost and feature richness will have to be found. Second, the desire to release the extended tool into the open source community also influences the decision taken with regard to the end user license: most commercial licenses do not allow redistribution of the program code. This leads to a preference towards extension of a community-backed open source solution, an option also favored with regard to the extendibility criterium; being able to modify the entire tool has its advantages over the limitation of building upon a provided API.

Choosing an open source solution does not narrow down license issues sufficiently, though. If, for instance, the tool used has been released under the GPL, the license used for derived software cannot place extra restrictions upon distribution terms. Because the extended testing tool is intended to be released as an open source project anyway, this is not really an issue; related software, such as the interface toolkits it is able to test and the applications tested with the tool, is not affected by these license terms. Nevertheless,

license terms for each tool should be studied to ensure any extension may be legally used and distributed.

## 4.2 Tool candidates

Having gathered and prioritized the list of requirements, a selection can be made of tools suiting this list. Candidates have been filtered on their availability for use in a project such as this; most commercial products have therefore been left out due to their lack of (legal) extendibility, and more importantly, their purchase cost.

### 4.2.1 Badboy

One of the commercial products considered, Badboy[1] is a closed-source application which is able to automate testing for Microsoft Internet Explorer. These aspects are the most important disadvantages of this tool; its extendibility and browser support are limited. However, Badboy does provide a script recorder and automation of test execution by providing command-line options to start a selected test suite automatically. Also, feedback from test runs may be requested by including special actions in test scripts, which are able to save log files to disk or forward messages by email.

Badboy Software offers another tool, named Wave Test Manager, as a controller to Badboy. WTM allows users to trigger execution of test cases on several computers at a time, which enables simultaneous testing on different platforms — although the use of this feature is questionable, as Badboy will only be able to test different versions of Internet Explorer at the same time. The test manager uses a Web-based interface, which allows for centralized maintenance; also, WTM aids in integration of test execution with the development cycle.

Although the available features are nice, Badboy does not meet the requirements needed to extend it to a component-based testing tool. License terms are unfavorable as well, a paid license being needed to use the tool within organizations having more than five users, and redistribution of the tool itself being prohibited — no mention is made of redistribution of extensions to the tool, although it is assumed this is legal.

### 4.2.2 SWExplorerAutomation

The other commercial tool included in this inventory is SWExplorerAutomation[2] by Webius. It is similar to Badboy, also solely supporting Microsoft Internet Explorer and being closed-source. Its license prohibits redistribution, except for included libraries, which is a good opportunity to use SWEA as a basis for extension. SWEA features a script recorder, which is able to convert scripts to C# or VB.NET; this balance between ease of use and the power of creating test cases through a true programming language is a desireable feature. This integration with the .NET framework is extended by providing support for NUnit test cases, which is great if one is developing in a .NET-based environment. However, because

[1] http://badboy.com.au/
[2] http://webiussoft.com/

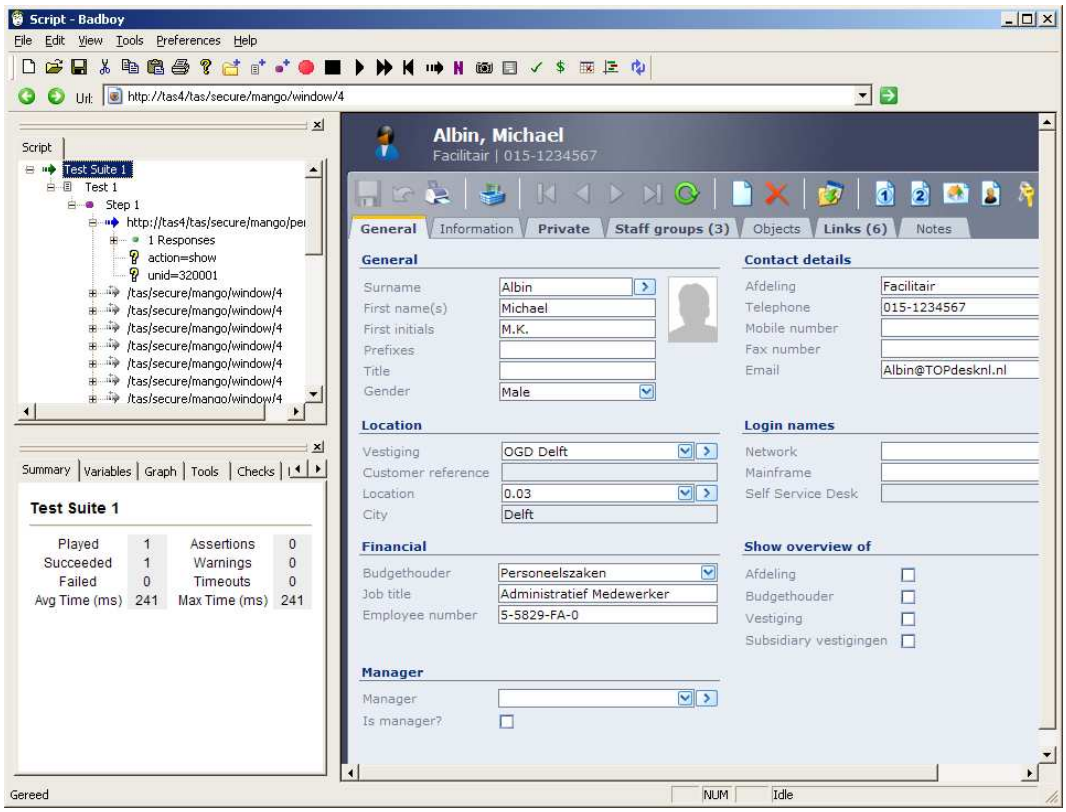


Figure 4.1: Recording interaction with a Mango card in Badboy

TOPdesk develops its software in Java, choosing SWEA as a basis for their testing tool would be irrational.

### 4.2.3 HtmlUnit-based tools

HtmlUnit[3] is an open-source project released under an ”Apache-style license”, which is very permissive with regard to redistribution and modification of code. It consists of a Java API to a browser emulator, written specifically for purposes needing automated browser behavior. Although it includes the Mozilla Rhino JavaScript engine to provide JavaScript support, it has been argued before that emulating a browser will never find bugs introduced by real browser quirks. The HtmlUnit developers also admit that complex JavaScript-based applications will probably not function correctly in HtmlUnit, making this an undesirable candidate for extension.

HtmlUnit is not a testing tool per se, although its Java API makes it easy to integrate into other testing frameworks on a variety of platforms. Tools such as Canoo WebTest[4] and

[3] `http://htmlunit.sourceforge.net/`
[4] `http://webtest.canoo.com/`

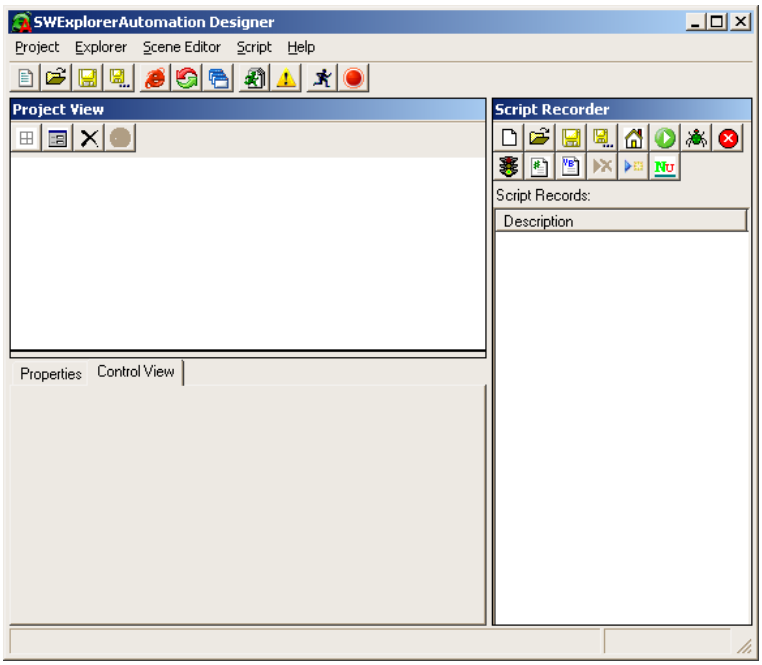


Figure 4.2: SWEA's recording interface

JWebUnit[5] use HtmlUnit for browser automation; the latter has been relicensed under the GPL. Both tools take advantage of other existing Java tools: WebTest uses a script format compatible with Apache Ant, while JWebUnit has been built as an extension to JUnit. They are also similar with regard to assertion capabilities, providing content checking functions found in nearly every other tool. However, WebTest also comes with a script recorder, which is able to create Ant scripts on the fly. Another advantage of WebTest over JWebUnit is the inclusion of more extensive reporting capabilities. Overall, the only downside to usage of a HtmlUnit-based solution is its emulation approach.

### 4.2.4 Watij

Watij[6] is a Java API to Microsoft Internet Explorer based on its Ruby equivalent Watir. It shares its advantages of being an open-source Java API with HtmlUnit-based tools; Watij has been released under the GPL, while Watij uses a license based on the BSD license. It is also limited to automation of a single, though real life, Web browser. Plans have been made to extend support to Mozilla Firefox and Apple's Safari; at the moment, ports of Watir are already available for this purpose. This flexibility makes it a good candidate for extension, but because of its lack of a simple scripting interface and the current implementation only having multiple browser support in Ruby, it is infeasible to implement Watij or Watir for

[5] http://jwebunit.sourceforge.net/
[6] http://watij.com/

use at TOPdesk's QA team. Watij also lacks a script recording interface; Watir and another derived project, WatiN, created in support of .Net development, do include a recorder.

An effort at developing an easy to use interface to Watir has already been attempted in Ruby and C# as WET[7], the Watir Extension Toolkit. WET developers claim to have improved the script recorder principle with their "proxied UI" mechanism. Their recorder does not monitor user actions performed in a browser window, but allows the user to specify actions to be executed in the browser by WET. In this manner, recorded test scripts always reflect the scenario intended by the end user. This approach is not as big a revolution as the developers claim it to be, though: using a regular script recorder and manually tweaking the draft version of the script through the recorder yields the same results as creating the entire script semi-manually. Also, WET does not solve the limitation of only supporting Internet Explorer; altogether, WET does not provide many additional features not found in Watij or Watir, which makes it one of the less likely choices for tool extension.

### 4.2.5 Selenium

As an open-source project with a large support base, Selenium[8] is a likely candidate for extension. This support base has also created many of the features required to adhere to TOPdesk's requirements, including integration with other testing frameworks, the availibility of both an easy to learn scripting language and several API's to other programming languages, and support for a wide range of browser and operating system platforms. These features have been implemented by creating several Selenium components, ranging from a test recorder extension for Mozilla Firefox to a test execution controller able to direct test runners for different platforms concurrently. All of these components have been released under the Apache 2.0 license, which is favorable in case the AJAX extension is to be released to the community. Java developers may benefit from Ant, JUnit and Maven integration provided by user-contributed extensions. This wide variety of components and integration options also make Selenium a very scalable solution. Another example of the flexibility of Selenium is its approach to communicating feedback; test results are posted to a Web page defined by the user, which is then able to further process this data.

One of the few downsides to Selenium stems from its JavaScript-based nature, combined with cross-site scripting security included in any decent Web browser. It necessitates the inclusion of Selenium's code on the same Web server as the application under test, or at least a simulation of this setup by introducing a proxy which is able to trick the browser into believing Selenium is running on the same server. The fact that Selenium is largely based on JavaScript also influences test execution speed, which may be an issue when an application's test suite becomes very large.

[7] `http://wet.qantom.org/`
[8] `http://selenium.openqa.org/`

### 4.2.6 Windmill

Windmill[9] is a Web test automation project, written by a group of developers who were dissatisfied with Selenium. Its implementation in Python consists of a script recording IDE, a command shell, and several API's, developed to write test scripts in Python, JavaScript or JSON notation. The IDE satisfies the requirement of having an easy to use script writing interface. The API's satisfy the requirement for inclusion of a simple test definition language, alongside a complete programming language. Its extension API is somewhat limited compared to that of Selenium; at the moment, it is only possible to add custom assertions. However, Windmill being an open source tool, it may be modified and extended in any way with a bit more effort.

A big plus of Windmill is its support for Internet Explorer, Firefox and Safari, one of the most important requirements set in tool selection. Another requirement, integration in the application's build cycle, is currently implemented by providing a command-line interface to execute a test suite unsupervised. Windmill has been released under the Apache 2.0 license, a favorable option, considering extension release. However, at the moment, the project is still very young, and this immaturity makes it less desirable to implement in a production environment.

## 4.3 Limitations of existing tools

### 4.3.1 Recognition of user events

Most of the tools available follow the browser controller approach, either through a browser plugin or an embedded browser. While they all implement playback of predefined scripts, only a small subset includes a script recorder; others rely on the user to completely write their test scripts himself. Even the most promising tools only support scripted JavaScript interaction in its most basic form: the execution of `onclick` events. Their focus is mainly set on filling out basic forms and checking for dead hyperlinks.

On an AJAX page, however, users generally trigger more events than are currently captured by any tool. This is caused by the fact that tools assume interaction only with certain elements, such as hyperlinks and form inputs. In an AJAX page, each element is theoretically an event source, from layout elements like `span` and `div` to the entire page body. Although these events are not captured, they could just as easily be, by recording events executed by the browser's JavaScript engine. Afterwards, they could be regenerated when executing a test case.

### 4.3.2 Recognition of page events

Classic Web pages basically trigger only one event not directly initiated by user activity, which is the `onload` event. Support for capturing this event is therefore included in most testing tools; however, AJAX Web pages have been built specifically on the concept of loading a page just once, and using lightweight interaction with the Web server to exchange

[9] `http://windmill.osafoundation.org/`

data from that point onwards. Although this interaction might be faster than loading an entire page for each request, there is still some delay involved, and `onload` style events should be recognized by an AJAX testing tool.

Most of this interaction stems from the usage of the `XMLHttpRequest` object, which has its own event model based on the `onReadyStateChange` event, the ideal candidate for notifying tools when a request has been completed. Another cause of delay, overlooked by testing tools, although heavily used even before the AJAX era, is the JavaScript collection of timer functions. These functions are used, for instance, in rollover style navigation menus; since navigation elements are essential in any application, there should definitely be some support for delays introduced by menu scripts.

### 4.3.3 Recognition of and interaction with custom UI elements

The most important aspects of user interface testing are executing actions on interface elements and inspecting their state. To be able to write and run tests, there needs to be some method of identifying interface elements, whether on generated code level in order to execute a test, or on interface definition level in order to write a test. Within HTML, the DOM and JavaScript, there are several identification methods available; either a specific tag attribute is added to HTML elements representing UI controls, or JavaScript object references are added to the `window` object. This attribute might be the standard `id` or `name` attribute, or a custom attribute unsupported by the HTML standard, in most cases silently ignored by the browser after being added to the page DOM. Both the HTML attribute and JavaScript option would make accurate widget references possible using very little code, although some additions to regular code generation by the interface toolkit would be necessary. This is actually an advantage compared to classic application testing tools, which usually lack the convenience of id's in element definitions.

To provide scripted interaction, components should expose their interaction methods to the testing tool. In most cases this will be trivial, due to JavaScript's lack of member access control. However, toolkit developers should also keep this into account, by making sure object methods and attributes are sufficiently accessible without the need for code unavailable to the test executor.

In a broader sense, a similar concept has been introduced as "built-in test" (BIT) wrappers [5]. These are code snippets added to components for testing purposes and may be removed before executing the final application build. As an addition, these snippets may include extra pre- and post-conditions for method calls, something not absolutely necessary in the AJAX testing tool. In fact, including testing code into the components themselves may introduce unwanted clutter. Keeping in mind the amount of referencing needed by the interface itself, this referencing code is probably already present in most interface toolkits. Also, automatically generating these wrappers as a HTML container element will be possible in some cases, alleviating the need for a developer to manually insert code to support testing.

In some cases, it might be possible to extract identification information from the interface definition files. Since user interface toolkits do not use a unified method of representing user interface designs and availability of these designs while testing is not guaranteed, this

method is more of a backup when identification through generated code is not possible.

Another question pops up from this analysis: does the interface toolkit need to conform to the testing application's rules, or is it the other way around? The answer lies somewhere in the middle: the interface toolkit needs to provide enough hooks for the test suite to be able to automate interface events and read interface states, while the test suite needs some way of knowing about these hooks. Due to different interpretations and implementations of UI controls between toolkits, there is no way of creating a unified testing tool; there will always be toolkits unable to fit into the tool's framework.

## 4.4 Selection

Summarizing the list of evaluated tools, one can see most are very similar in their approach of test automation. Even tools that are being developed by the open source community have many features in common, which raises the question of why these communities do not unify their efforts at a single UI test automation suite. Most of these open source tools are therefore limited in their features, due to their small support base; however, Selenium sets itself apart from this group, supporting a wide range of features and tool components.

Then, there are tools developed as commercial products. These tools tend to be more complete in their feature set, which is obviously necessary in order to be commercially successful, but they restrict other developers in their ability to write extensions such as a component-based testing tool. Their license terms are also unfavorable in case of release of the extension as an open source project. This basically eliminates any commercial solutions from tool selection.

Dividing the list of tools by another criterium, browser support, one finds that most tools support a single browser. Unsurprisingly, this browser happens to be the one with the largest market share by far: Microsoft Internet Explorer. While testing applications in MSIE is important, testing just for this browser does not meet the requirements set. The emulation-based tools are also unsuitable for AJAX testing, because they introduce their own browser engine, ignoring real browsers. This leaves three testing frameworks: Watir, Selenium, and Windmill. Although the first claims additional support for Mozilla Firefox and Apple's Safari, this support has been implemented as separate ports. Selenium's browser support[10], which extends beyond the three implementations of Watir, is largely achieved by integrated pieces of JavaScript-code, which is easier to maintain than separate tools. Windmill claims to support the same browsers as Watir does, but in one integrated solution. Additionally, Watir is based on Ruby, and Windmill has been implemented in Python, while Selenium is Java- and JavaScript-oriented; because TOPdesk's developers generally know a lot more about Java and JavaScript than Ruby or Python, Selenium is again favored over other solutions. Finally, Selenium's maturity is favored over the relatively new Windmill.

In practice, the requirement of test integration with the application's build cycle is interpreted differently by each tool. Some tools provide nothing more than a command-line in-

[10]It should be noted that Squish, a commercial tool priced at around 2500 EUR, also supports a list of browsers comparable to that of Selenium.

terface to start unsupervised test execution, others integrate into an existing automated build environment. For those tools implemented as an automation library rather than a stand-alone application, integration is less of an issue due to the shift of responsibility for integration to the test developer, who will have to create his own framework for the test suite to execute in. Including testing in the application's continuous integration, both procedurally and technically, is preferred where available; this approach keeps execution scheduling and feedback at a central location. However, because of the lack of a standardized solution for continuous integration, testing tools do not include integration with any of these solutions; integration is instead provided by third-party extensions. Selecting a testing tool based on continuous integration support is therefore best done by availability of these extensions; this makes the tool with the largest active developer community the most desireable candidate.

Feedback features are also subject to broad interpretation. As with continuous integration, responsibility for feedback methods in library-based tools is left at the test developer. Stand-alone tools tend to support generation of detailed log files, which may be processed by the continuous integration framework, saved to disk, or emailed automatically. Some tools allow the user to specify extra logging moments at intermediate moments by including special actions in test scripts, others process logs only when test execution has finished. Requirements for the AJAX tool do not enforce usage of any one of these methods; the list of tool candidates is therefore not narrowed down by this criterium.

All in all, the set of requirements combined with available tools points at the most versatile and arguably most widely supported Web interface testing framework: Selenium. These criteria, along with the fact that a robust AJAX user interface testing framework for a specific toolkit — TIBCO's GITAK[11] — has been developed upon Selenium before, make this seem the most suitable choice for tool extension.

[11] `http://www.tibco.com/devnet/gi/product_resources_gitak1.jsp`

# Chapter 5

# Selenium

Selenium is a testing tool collection made up of several components. Each of these is built upon the same engine, Selenium Core, implemented in JavaScript. To this project, the most interesting components are Selenium Core and Selenium IDE.

## 5.1 Features

### 5.1.1 Selenium Core

Selenium Core forms the test execution engine used in all other Selenium components. Also, it includes a test runner framework, capable of executing Selenium tests from within a Web browser window. Like most Selenium components, Core supports so-called ”Selenese” script syntax, a simple script definition language. Because Selenese is limited to definition of a list of sequential interactions and assertions, it is not suitable for scripts needing branches or loops. This limitation may be circumvented by using Selenium RC (Remote Control) which supports scripts written in higher level languages, allowing for more complex script constructs, while still using the same Core test engine. Selenium RC is also used to drive Selenium Grid, a solution aimed at developers who wish to run tests in parallel, while managing a heterogeneous set of platform configurations from a central location. Figure 5.4 shows a diagram describing these Selenium components.

As noted before, the test runner included in Core has another drawback: to circumvent a Web browser’s cross-browser scripting security, it needs to be installed on the same Web server as the application under test. This may be done directly, or virtually, by using a proxy server that tricks the browser into thinking the application and Core are running at the same site.

### 5.1.2 Selenium IDE

The IDE component of Selenium is developed to facilitate test suite maintenance. Although other Selenium components support a wide range of Web browsers, IDE has been written specifically as an extension to Mozilla Firefox. It includes a test recorder, which is able to build a test script through monitoring of application use within Firefox; one of this project’s

goals is to extend this recorder in order to be able to generate more comprehensive scripts for an AJAX environment. Also, Selenium IDE includes functionality to debug scripts, along with a script editor featuring autocompletion.

Most of the use cases defined earlier are already satisfied by current Selenium Core and IDE features. That is, creation and modification of test scripts can already be done in an intuitive way through IDE, just as manual execution of a test suite. Generation of a basis for a higher order language script from an existing Selenium script is also possible through an IDE export option; these scripts may be executed through Selenium RC. Finally, Core provides the means for scheduled test execution along with report generation.

## 5.2 Extension support

As all Selenium components are released under an Apache 2.0 license, their respective source code sets are freely modifiable in any way. Apart from this freedom, both Core and IDE provide intuitive extension interfaces. It is possible to define custom command handlers (Figure 5.1), element locators (Figure 5.2), and assertions (Figure 5.3), in a predefined extension file, which is executed by Core. Meanwhile, IDE is extendible by defining new strategies to use for action recognition while recording test scripts, which allows these custom handlers and locators to be automatically added to test scripts, where appropriate. These extension interfaces are an ideal platform for an AJAX extension; after all, TIBCO's testing tool GITAK has also been based upon the Selenium extension API.

```
Selenium.prototype.doTypeRepeated = function(locator, text) {
    // All locator-strategies are automatically handled by "findElement"
    var element = this.page().findElement(locator);

    // Create the text to type
    var valueToType = text + text;

    // Replace the element text with the new text
    this.page().replaceText(element, valueToType);
};
```

Figure 5.1: Adding a ”typeRepeated” action to Selenium (taken from Selenium documentation)

## 5.3 AJAX support

Like in every other available tool, no explicit AJAX support is currently implemented in Selenium. Although this term is loosely defined, some aspects that are clearly part of AJAX

```
// The "inDocument" is the document you are searching.
PageBot.prototype.locateElementByValueRepeated = function(text, inDocument) {
    // Create the text to search for
    var expectedValue = text + text;

    // Loop through all elements, looking for ones that have
    // a value === our expected value
    var allElements = inDocument.getElementsByTagName("*");
    for (var i = 0; i < allElements.length; i++) {
        var testElement = allElements[i];
        if (testElement.value && testElement.value === expectedValue) {
            return testElement;
        }
    }
    return null;
};
```

Figure 5.2: Adding a ”valuerepeated” locator to Selenium (taken from Selenium documentation)

```
Selenium.prototype.getTextLength = function(locator, text) {
    return this.getText(locator).length;
};
```

Figure 5.3: Adding an ”assertTextLength” action to Selenium (taken from Selenium documentation) — assert actions are automatically added for each defined ”get” method

are difficult to describe properly in Selenium test cases.

One of these aspects relates to the X in AJAX: awareness of its most common method of communicating XML data, i.e. XMLHttpRequests, in particular the ability to tell Selenium to wait for completion of such a request. Although a command named `waitForPageToLoad` is available, this waiting only applies to waiting for an `onload` event associated with an entire page, not incremental page modification done by AJAX code.

Support for JavaScript is also still lacking, particularly in Selenium IDE's recorder feature, when measured by AJAX standards. For instance, a DOM element might contain both an `onmousedown` and `onmouseup` event, alongside an `onclick` event. When recording a script, using the mouse to click such an element will only record the `click` command, ignoring events triggered by pressing and releasing the mouse button.

## 5.4 Project extensions

Extending Selenium will consist primarily of the introduction of the concept of rich interface components to Core and IDE. While Selenium's extension API will probably suffice for the largest part, the necessity to modify other parts of Selenium should be kept in mind. It should also be expected that Mango will need some adjustments in order to improve robustness and stability of the extension.

Modifications to Core and IDE can be roughly divided into four steps: defining new locator strategies for identification of Mango components, adding commands for interaction with Mango components, adding assertions for inspection of Mango components, and adding strategies for recognition of Mango components by IDE. Implementing these modifications in this order seems the most natural, as they are all somewhat dependent on their predecessors.

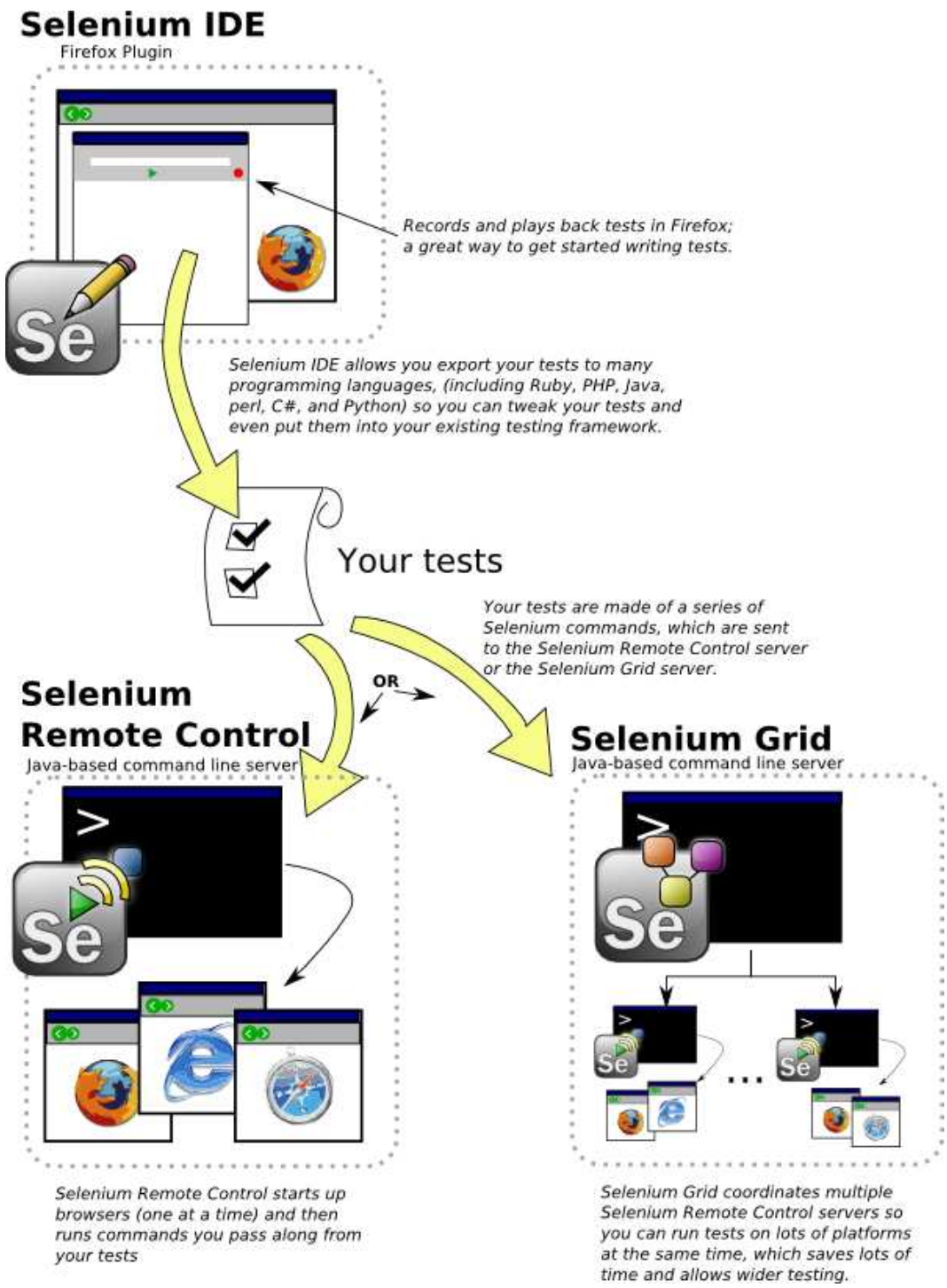


Figure 5.4: Selenium's components, as explained on the Selenium Web site

# Chapter 6

# Tool implementation for Mango

After the testing tool platform to build upon has been chosen, implementation of the tool extension nears. The next step towards implementation is a dissection of Mango, in order to discover the best way to have it disclose its details to the Selenium extension. To reflect differences between Selenium and the extension, its prototype has been dubbed Arsenic, to keep in line with naming the tool after a chemical element.

## 6.1 Structure and hierarchy of Mango

Mango is not just a Web interface construction toolkit: it is a platform independent GUI toolkit, providing code for several aspects of GUI implementation. Layers have been created to represent these aspects: there is a layer of models, providing the interface logic built upon application logic; a layer of controls, which are abstract representations of visual interface components; and finally a layer of peers, translating the abstract controls to platform-specific widgets.

### 6.1.1 Models

Models represent the internal state of an object represented in a Mango GUI, tracking aspects such as data value and writeability. They form a link between application logic and the actual user interface; as such, a thin line exists between what should be considered interface logic implemented in models and application logic implemented deeper in the application code.

To an interface testing tool, this layer of logic is primarily one to validate, rather than retrieve information from. However, its connection to the application's data layer is valuable in providing reference points to the tool. Having unique and consistent identifiers for controls at the DOM level is essential in ensuring tool robustness; models can aid this aspect by disclosing information to the control layer about, for instance, database field names related to a text input control, or resource keys related to a labeled button. This identification method does not solve all identifier issues, though: a text input does not necessarily correspond to a field in the application's database, and a button control may be displayed as an image instead of a button containing a descriptive text.

### 6.1.2 Controls

Controls represent the first visual representation of the GUI. Although they describe what the control comprises, including interaction methods and its reactions to model changes, controls do not specify layout or implementation details of their rendered counterpart. This aspect is left to peers, the platform-specific control implementation layer.

To the testing tool, the most interesting aspect of controls is the fact that they specify the visual hierarchy of the GUI. More specifically, the tool will be able to capture and replay sensible scripts only if it knows about the relationship between a control and the interface element it corresponds with. Although there may seem to be a trivial solution to this problem, finding out the control a triggered event logically belongs to is certainly not trivial. For instance, Mango contains a control to select a value from a predefined list, using a text input to display the currently selected value, and an image button, which can be clicked to show a list of possible control values. In a GUI, elements representing this control are part of a larger hierarchy of container elements. The testing tool cannot easily determine which elements are ”relevant” to event capturing, and which elements are merely introduced for layout or other grouping purposes. To this end, it should be decided at control level which controls are responsible for each event triggered at the interface level.

### 6.1.3 Peers

Peers form the final layer to Mango at the visual end; a set of peers consists of control representations specific to a certain platform. In this case, the only relevant set is WebMango, Mango’s HTML-based peers used in rendering the main user interface of TOPdesk. These peers are responsible for generation of the HTML elements corresponding to controls available in Mango. As such, they create a direct link between Mango and Arsenic, useful in passing identifier data to the latter.

An issue that pops up in the translation of controls to peers is the fact that the conceptual controls, as added to the application screen, do not necessarily correspond one-to-one to HTML elements in the generated DOM tree. This discrepancy stems from the ability of a control to add other controls to its container parent, leaving itself out of the actual GUI tree. This means that, whenever a relevant control — as defined in the above section on controls — needs to be found for a certain peer, this search is not as trivial as traversing up the tree to find the desired control. An example of this is again the multiple choice selection control: when a control instance of this type is added to an interface, the actual selection control is not added to its parent. Instead, a generic container control is added, containing a text box and an image button. To find the selection control to which these controls are added, one would need to traverse up to the container control, which contains a reference to the selection control. These nontrivial paths to find the relevant control should be kept into account in the design of the identifier mechanism.

## 6.2 Connecting Mango to Arsenic

With Mango's internal layout explained, an implementation approach for enabling Arsenic to use relevant portions of these internals needs to be determined. These approaches can be divided into two categories: one based on Mango disclosing a fixed set of information along with the generated GUI, the other relying on Arsenic querying Mango on control information as test generation or execution progresses.

The relatively most straightforward approach is the inclusion of additional attributes to generated HTML tags, or perhaps the introduction of helper tags containing the identification data. These attributes and tags can be read and manipulated by Selenium like any DOM element, requiring little to no modification to Selenium for this purpose. On the other hand, using this approach, peers, controls and models themselves would need to be modified to propagate identifier data to HTML attributes; changes would have to be made throughout Mango's source code, which could interfere with other Mango functionality. Additionally, providing a fixed set of attributes yields a static solution, leaving Arsenic and test script writers unable to request control details not provided in these attributes.

A more dynamic approach lies in the implementation of a separate query interface, for Arsenic to use whenever control details are needed. This interface would need to communicate with Mango directly and therefore be available on the Web server alongside WebMango, to be able to query the live GUI controls in Java for information on generated identification and layout passed to the DOM. An advantage of this solution is the fact that Mango itself does not need to be modified — perhaps slightly, but these changes would interfere a lot less than the previous approach. The flip side is the complexity involved in Selenium modifications: Selenium is not designed to send additional server requests for identification purposes while recording or executing a test script. Additionally, execution of the test suite would slow down considerably when identification requests need to be done with each execution step.

These two approaches lead to the idea of a hybrid solution: one that minimizes interference with existing Mango code, while also limiting the increase of load on Arsenic. A solution of this type could involve a query interface in JavaScript on the client side, which keeps its GUI identification data updated with each page load and update. Because of its JavaScript nature, it would cooperate well with Arsenic, and because its client side data is kept synchronized during updates of the page DOM, server roundtrips to fetch identification data would be minimal compared to a server side query interface. Its interference with existing Mango code would be limited to the introduction of a hook, called at each interface update. The downside of having a static set of identification methods, due to the data provided by the query interface, remains unchanged; identification data is just moved from the DOM to JavaScript. Also, in practice, revealing a toolkit's internals to a testing tool without modifying these internals is a fairly utopic situation, because these internals tend to be hidden from external code. These arguments make this approach not particularly favorable over the two previously mentioned; the approach adding DOM attributes will therefore be implemented in Arsenic.

### 6.2.1 Limitations of Selenium

As Selenium has been designed as a tool for testing HTML-based Web sites and applications using relatively little client-side scripted automation, it has some limitations for which there may or may not be a graceful workaround. One of these limitations is the assumption that waiting for page content to load should only be done when navigating to a new URL, for example by clicking a link or submitting a form. Asynchronous communication with the Web server and incremental updates are not taken into account directly by its test runner. Fortunately, the execution engine provides an extension hook named `terminationCondition`, allowing an extension developer to define the condition that needs to be satisfied before a script command may be considered completed. Combining this feature with code monitoring the status of outstanding `XMLHttpRequest`s, it is possible to define termination conditions waiting for these requests to complete.

There are still some drawbacks to this method. First of all, it should be noted that integrating this functionality into Arsenic will remove obvious methods for testing interface consistency in cases where the end user does not wait before executing his next interaction. Also, some highly dynamic interfaces may never enter the idle state sought by Arsenic; a continuous stream of new `XMLHttpRequest`s prevents it from knowing whether interface updates related to a certain event are finished. This issue could be circumvented by extending the interface toolkit in question to supply this information directly, but this is by no means a trivial task, possibly requiring major modifications to the toolkit.

Another fundamental limitation of Selenium is its scripting syntax, which consists of a list of commands, each formed by an action, a target and a value, as shown in Table 6.1. While this syntax suffices for some scripting purposes, several commands require a more flexible solution than a fixed set of three text fields — especially more complex interaction methods defined for Mango interfaces. For instance, recognition of interface elements will sometimes require more than one criterium, which is difficult to do when using just one text field for specifying a command target. This is particularly noticeable when attempting to identify identical elements occurring several times in a page; an example of this issue as it appears in TOPdesk is shown in Figure 6.1. Except for their XPath locators, which are difficult to construct in a robust manner and therefore prone to failure whenever a change is made to the DOM, the icons associated with both text areas are not discernible through a Selenium identification method. Ideally, one should be able to enter several criteria in defining this target, resulting in an identifier describing ”the stamp button belonging to the Request field”, which is both robust and easy to understand for a human reader. Unfortunately, since this syntax is a core feature of Selenium, extensions need to work around these limitations by overloading the three available fields.

In Mango, using multiple target criteria will also be useful in minimizing script writing errors. A verification is needed in order to ensure that the action to be executed consists of operations supported by the receiving control. To this end, along with the action itself, its target identification, and possibly a value, the control type should be included in the command definition. Lacking an extensible number of fields, this means that one of the existing fields should contain the control type name. Because the contents of the ”value” field is not commonly bound by any syntactical restrictions, this is the least likely candidate

| Command | Target | Value |
|---|---|---|
| open | / | |
| type | q | selenium |
| clickAndWait | btnG | |
| clickAndWait | link=Selenium web application testing system | |
| assertTitle | Selenium web application testing system | |

Table 6.1: A simple Selenium script, demonstrating its syntax

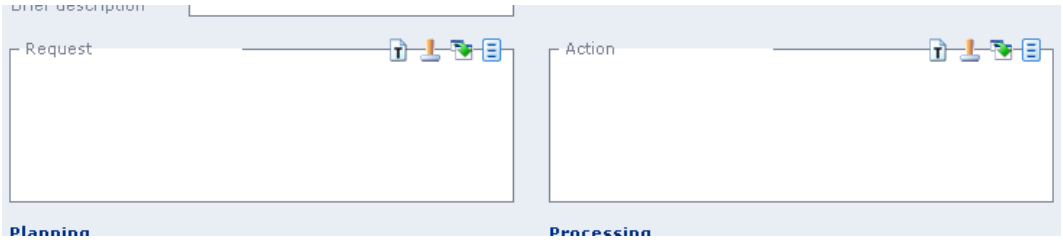


Figure 6.1: ”Memo fields” (text areas) having multiple identical icons

for control type inclusion, as it would also involve executing additional parsing operations. Using the ”target” field is a more suitable candidate, as this field is already used for control identification purposes. However, Selenium generally restricts target identification syntax to a name-value pair containing an identification method and a single parameter. Extending this field would mean including the control type in one of this pair’s elements. Either way, each existing identification method would require modification to support identification by the additional control type criterium. This leaves the ”action” field as the most suitable option: because operations supported by each control type will need to be defined anyway, generating different actions for each type is a logical solution.

A similar solution may be used in identifying controls that are part of a larger composition, such as the icons belonging to the special text area referenced earlier. Ideally, these controls contain references to the composition to which they belong. Vice versa, in defining operations available to a particular composite control, one will also need to specify the controls making up the composition. In doing so, additional actions may be generated referencing these children as subcontrols of a particular control type. This is a benefit of the modified script syntax in script execution; adding subcontrol information to separate actions allows the script runner to identify which element of a control needs to be targeted in event invocation.

The downside to this solution stems from the fact that target identification is done independently from action execution. This means the control type included in the action cannot be used to disambiguate between multiple controls using the same identifier. Use of the control type in the action name as target identification is therefore limited to validation purposes, throwing an error whenever an expected control type does not match the actual control type.

The static nature of Selenium's command syntax also prevents the use of dynamic discovery of control events. An early idea for extension implementation involved monitoring of JavaScript events triggered by the user, recording the interaction executed to trigger the event. So, instead of defining interactions available for each control manually in advance, Arsenic would detect whether a given event has any effect in combination with a certain control type, and automatically define an action accordingly. However, because Selenium needs to have a complete list of available actions and locators before starting test recording and execution, such a solution is impossible to implement without including every event known to Web browsers to every control known to the interface toolkit in Arsenic's arsenal of actions. Although this is theoretically possible, it would also generate nonsensical actions such as typing into buttons and check boxes, which makes this solution unfavorable.

### 6.2.2 Referencing Mango controls

Although much has already been said about identifying Mango controls, no mention has been made of the identification data already present at the Web interface level. Until the introduction of Arsenic, the only code needing references to the HTML controls generated by Mango was Mango itself. Therefore, a numeric identifier is generated by WebMango the instant a control is added to the interface. This identifier does not necessarily remain the same between interface instances, rendering it unusable for a tool that needs to reproduce the same interface operations more than once. So, in order to be able to consistently identify an interface control, the identification data generated by the server needs extension.

First, data and presentation of the additional identification mechanism need to be determined. The presentation issue has a fairly trivial solution; browsers allow HTML elements to contain any custom attribute, a feature which is already exploited by Mango for other purposes. Adding the identifier as an element's attribute therefore makes for a practical and elegant solution.

Picking a unique and preferably relevant identifier for each control present in the interface is a far less trivial task. The most intuitive approach for both tool developer and test case developer would be to use a string representing the purpose of a particular control. However, having the application developer define such a string for each control instance is a tedious task, not to mention a procedure that would introduce unwanted clutter to the application code.

This is where Mango's architecture benefits identifier generation: as noted before, each Mango control is backed by a model, defining the control's relation to the application layer below the interface. So by attaching a model to a control instance, the application developer already defines the control's purpose. Extracting specific model data, which has been marked as identifying a particular model instance by a Mango developer, from the control as a text string will therefore yield a suitable identifier to add to a corresponding attribute at the Mango peer level. This approach requires only modifications to Mango itself, instead of needing additional input from application developers. The issue of identifying subcontrols within composites can be solved in a similar fashion: while these are usually not backed by very descriptive models, a slight modification to the control's code adds a manually specified descriptor, referencing the subcontrol's role within the composite and its parent's

identifier. This leaves only stand-alone controls using nondescriptive models to deal with: while these controls are usually not the ones interacted with by the user — for instance, container controls defining page layout — there should be an option for developers to add an identifier to such a control. Therefore, a method for overriding the identifier generated by the control and model itself has been added to the base control class in Mango, to enable overrides for specific control instances.

To deal with controls uniquely identifiable within another identifiable section, but not within the entire GUI, another locator has been added to Arsenic; this locator follows an identifier path from the root of the interface into its hierarchy, narrowing down the level of uniqueness a control needs. As long as a control is placed in an uniquely identifiable container, its own identifier needs only be unique within this container.

Because Mango peer generation tends to follow a relatively straightforward pattern, sharing control information with Arsenic may be done using fairly compact definitions. An example of these definitions, as added to Arsenic's modified Core component, is illustrated in Figure 6.5, which shows a mapping, comprised of a (subset of) a list of control types available in Mango. This mapping contains data on interaction methods possible, and in case of a composite control type, a list of subcontrols owned by the control. Interaction methods correspond to JavaScript events, while subcontrol names such as `contextbutton` are also explicitly defined in the control itself. At run time, this control map is converted to a list of actions usable by Arsenic. This turns out to be a fairly practical solution, which requires minimal effort when adding new control types, and not just for Mango, as other toolkits have a lot in common with regard to control layout at the HTML level.

### 6.2.3 Verifying test results

Besides interacting with the user interface, any testing tool should also verify whether the actual interface state matches expectations, which are set according to requirements defined before and during application implementation. Automating definition of these verifications has been researched and implemented in several different ways: the most basic verification step obviously being to check whether a sequence of actions, as defined in a script, is able to execute at all [1] [18], which is trivial to automate and implicitly checked when executing a test. This type of verification predates AJAX interfaces, and was already useful when most errors where caused by code executed at the server side, or by common bugs like broken links; with AJAX interfaces, these verifications are still relevant. Another more or less trivial aspect of automatic verification consists of monitoring errors propagated by the application platform [11], which in this case means checking for DOM and scripting errors generated by the Web browser. These checks are fairly trivial to implement because of the fact that they are not specific to any Web application; more interesting aspects of verification are those related to automatically defining conditions that need to hold for individual applications. These conditions are particularly interesting to complex database-driven Web applications built upon a control toolkit, which are very dynamic in nature, as opposed to AJAX websites presenting semi-static data in a dynamic way; many interface tests will verify whether presented data is correct for a specific set of input data, especially when dealing with corner cases known to cause problems.

Several solutions found for the application-specific verification problem require a certain amount of guessing to come up with a suitable oracle; it is up to the user to tweak these conditions. Other automated testing solutions avoid the automatic verification issue by requiring the user to define all conditions manually; in regular record-and-playback, this approach is also used. Selenium already supports verification actions on element attributes such as `value` and `checked`, as well as checks to verify whether an element or a specific piece of text is present in the DOM. While this aspect of testing differs from regular HTML testing, the fact that most of a Mango control's properties are represented by regular HTML attributes makes extension implementation fairly straightforward with this regard, because most of these existing verification actions can be reused. In this case, elements needing attention are so-called "skinned" controls, like check boxes and radio buttons represented by images rather than their platform-native counterparts, as illustrated in Figures 6.2 and 6.3. While these contain an attribute specifying whether the control is in an "activated" state, a property to be checked often during testing, this attribute is not the `checked` attribute Arsenic expects, which means additional verification actions should be defined.

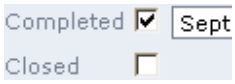


Figure 6.2: A check box implemented as a native widget

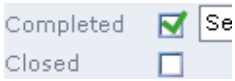


Figure 6.3: A check box implemented as a set of images representing different states

A point of attention with regard to the insertion of verification steps in test scripts is the fact that Selenium's script recorder interface provides additions to Firefox context menus. These additions generate a list of relevant verification commands for the selected element; for instance, when the context menu is called over a text box, it provides a menu item for adding a command to the loaded Selenium script verifying the value of the text box to be equal to its current value. A useful extension here is the addition of context menu items relevant to the selected Mango control, in order to allow users to add verifications in an intuitive way.

Automation of the verification aspect could be obtained by having Arsenic insert verifications whenever it notices a DOM modification following a user action in the script recording phase; this DOM modification should then be translated to an application interface state change in terms of components. This is by no means a trivial task: first of all, the relevance of an individual DOM modification in an interface change needs to be considered, where some modifications may be irrelevant altogether, while others form a single change when combined. DOM modifications will only be part of what users want to verify: sometimes, one wants to check whether an element in the interface has not changed at all. Also, the point at which verifications should be inserted has to be determined; a

user recording a script might not be interested in a particular inserted verification, which makes suggesting verifications, instead of immediate insertion, a more attractive solution. Arsenic's implementation does exactly this: suggestions are shown in its recording interface (Figure 6.4), whenever a verification is assumed useful for insertion. Usefulness is determined by monitoring DOM changes; whenever insertion or deletion of a Mango control is detected, an assertion checking this fact is suggested, and when an attribute belonging to a Mango control is modified, the control type definition (Figure 6.5, the `attributes` property) is consulted to check whether a particular attribute is deemed relevant within context of the control affected.

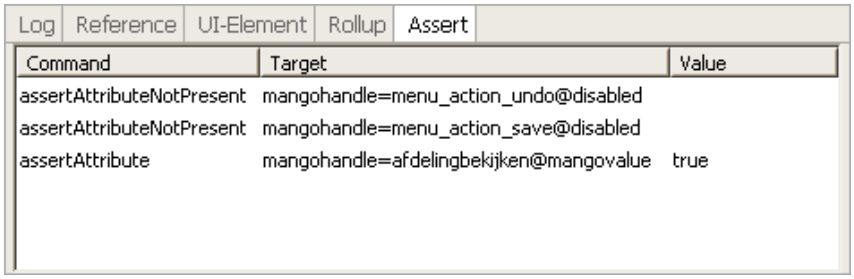


Figure 6.4: Suggestions generated by Arsenic IDE

## 6.3 In practice: TOPdesk

Implementation of a Mango testing tool requires a meta-testing platform, which in this case is the most well-known and used application using Mango code: the service management software it was designed for, TOPdesk. At the time of implementation, TOPdesk developers were in the process of converting some application modules to Mango, while the majority of the software core and other modules retained their legacy code. This limited practical testing opportunities to isolated module interfaces, as interaction between modules would require extensions to Arsenic unrelated to Mango. Some effort has been made to bridge these gaps in extension code, but focus has been on improving automated testing for the WebMango interfaces themselves.

The first few test runs revealed a practical solution to the problem of waiting for asynchronous requests to finish. WebMango includes a connection handler at the client side, written in JavaScript, to process requests and responses belonging to `XMLHttpRequest`s. By hooking into these processing procedures, Arsenic is able to monitor the number of outstanding calls at any time. Requiring this number to be zero as a termination condition for each Mango-related operation defined in a test script ensured correct timing in execution of subsequent commands.

This solution functioned perfectly in all but one situation encountered in TOPdesk card testing: the list of options in the custom multiple choice selection control. While most interface updates in WebMango are executed directly from a call's response, the so-called grids involved in the selection control are populated by JavaScript after the response has

```
var controlTypes =
{
    "org.mangosdk.control.misc.Button":
    {
        attributes: ["disabled"],
        actions: ["click"]
    },
    "org.mangosdk.control.image.ImageButton":
    {
        attributes: ["disabled"],
        actions: ["click"]
    },
    "com.topdesk.mango.control.AbstractContextMenuControl":
    {
        children:
        {
            contextbutton: "org.mangosdk.control.image.ImageButton"
        }
    },
    "org.mangosdk.control.text.TextBox":
    {
        attributes: ["value", "readonly", "disabled"],
        supertype:
            "com.topdesk.mango.control.AbstractContextMenuControl",
        actions: ["click", "keyPress"]
    },
    "org.mangosdk.control.selection.ComboBox":
    {
        attributes: ["disabled"],
        supertype:
            "com.topdesk.mango.control.AbstractContextMenuControl",
        children:
        {
            textbox: "org.mangosdk.control.text.TextBox",
            dropdownbutton: "org.mangosdk.control.misc.Button"
        }
    }
};
```

Figure 6.5: An example control type definition, as used by Arsenic

been processed. This required a different termination condition for commands triggering the display of such a grid: to have all visible option values loaded on screen.


Figure 6.6: Typing text into the text-completing selection control

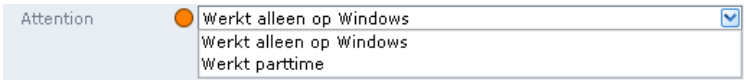


Figure 6.7: Opening the text-completing selection control option list

As mentioned earlier, TOPdesk's user interface has not yet been completely converted to a Mango-based implementation. Among these non-Mango elements is its main navigational interface, consisting of an HTML frameset with a JavaScript-based menu structure and a window manager allowing several TOPdesk pages to be open within the same frameset at once. The existence of this window manager is required to be able to navigate between TOPdesk's WebMango cards. Therefore, to be able to write test cases where navigation between cards is involved, support has been implemented for recognition of and interaction with TOPdesk's window manager.

An issue with uniqueness of the identifiers generated by model and control inspection was discovered when testing a date selection control, which includes a calendar to allow the user to select the desired date. The clickable day labels generated by the calendar code were not discarded after the calendar itself had been removed from view, but instead kept hidden by setting their height and width to zero. However, when a user requested another instance of the calendar, these labels were not reused, but a new set of labels was generated, using identifiers made up of the date a particular label represented. This conflicted with the old labels still in existence, causing the new labels to be indistinguishable from the old by Arsenic. A particularly ungraceful solution to this problem was ultimately required, which made sure invisible calendar labels were ignored when searching for an identifier.

Controls using a complex set of event handlers were also troublesome to interact with. In particular, the autocompleting selection control required interaction methods different from those normally generated by Selenium. One of these methods was the "type" action, inserted into a script whenever the user types some text into a text box, in this case the autocompletion box. Selenium's implementation of this action involves setting the value of the text box and triggering its `onchange` event. However, the autocompletion control expects to receive key press events for each key pressed by the user, in order to update displayed suggestions with each letter added to the text. This required replacement of recording whenever a text box value changed to recording of each key pressed, which fixed the autocompleter's behavior on text input.

After this issue was fixed, another one popped up regarding the selection of a suggestion from the list displayed below the text box. Because not all available suggestions are prefetched when typing, instead loading them on demand while scrolling through the suggestion list, Arsenic is not able to select a suggestion without first triggering the scroll events

leading to the display of the desired value. Support for recording and executing scroll events therefore needed to be implemented.

Most of the extension implementation and testing was done using just one TOPdesk card as a reference; only after the initial implementation phase were other TOPdesk Mango pages tested, to see if the extension design would scale well to other instances of Mango interfaces. Results were promising: the only new bugs found involved recognition of controls not used in the first card, and this issue was trivially solvable by adding these controls to the list of available Mango controls.

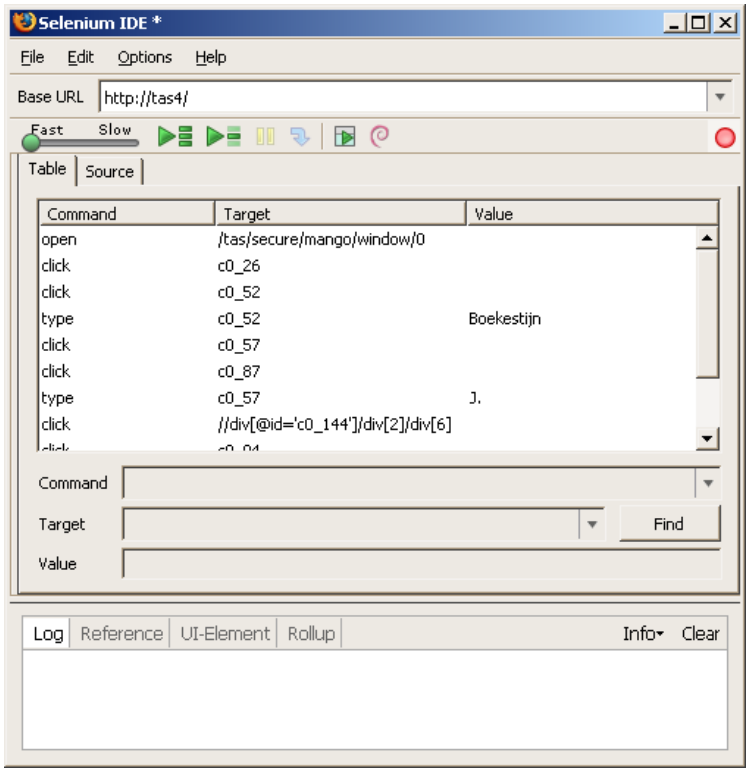


Figure 6.8: Recording interaction with a Mango card in Selenium IDE

## 6.4 A case study: Incident management

One of TOPdesk's core modules, incident management has recently been converted to Mango by its development team. This recent conversion makes it a nice candidate for a short case study to test both Arsenic IDE and the pages making up the incident management module. As a scenario, the modification of an incident is performed while Selenium IDE is recording steps taken; then, the same steps are taken while Arsenic IDE records its own version of the script. This scenario is intended to evaluate whether a script created through Arsenic IDE is more robust, while requiring less effort from the user.

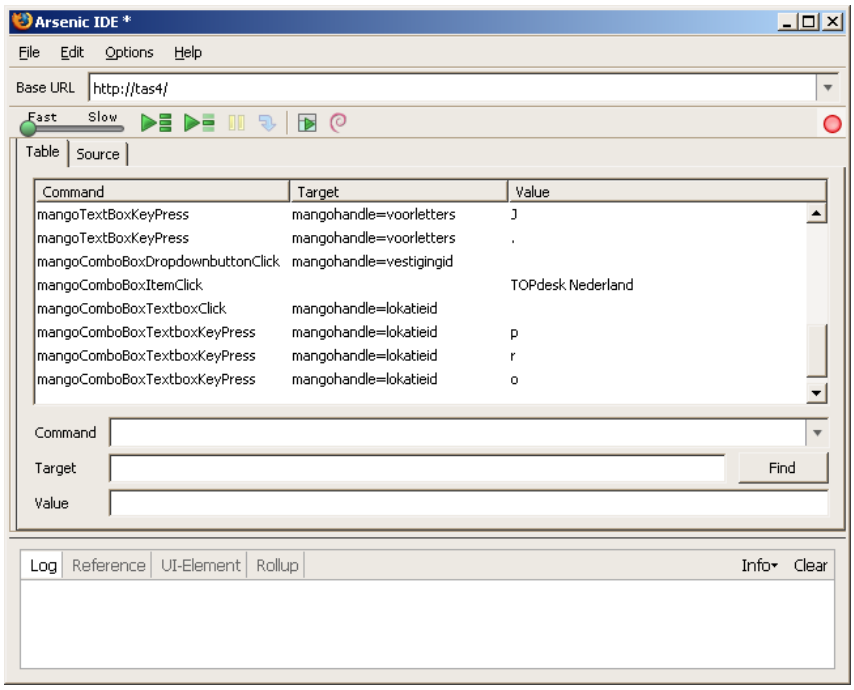


Figure 6.9: Recording interaction with a Mango card in Arsenic IDE

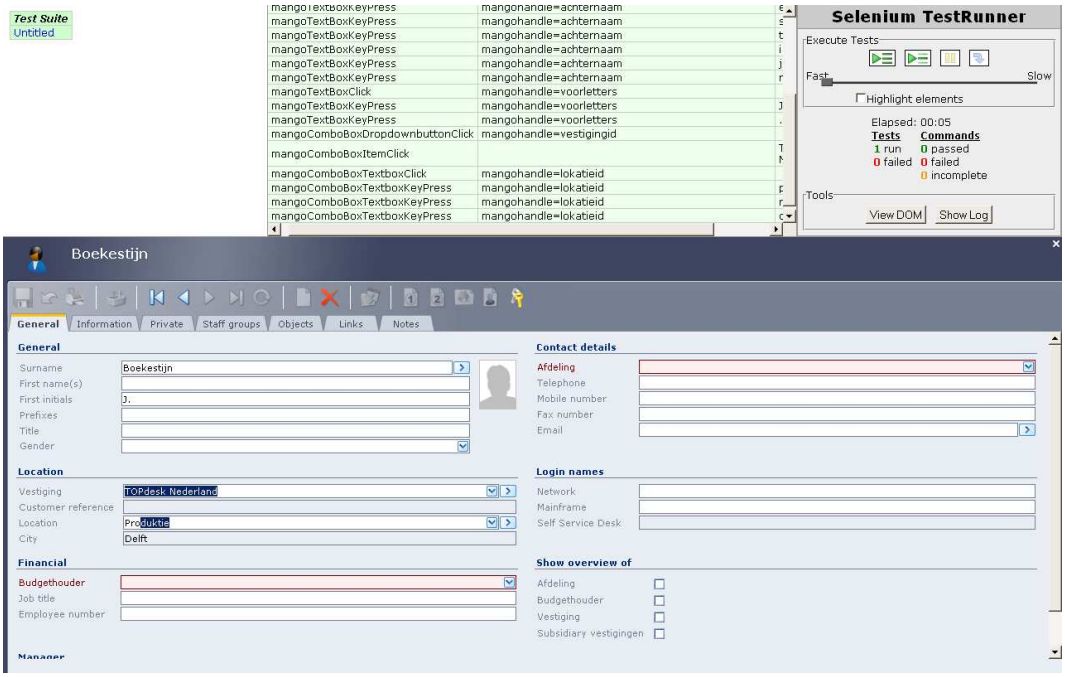


Figure 6.10: Selenium's TestRunner interface, running a Mango test

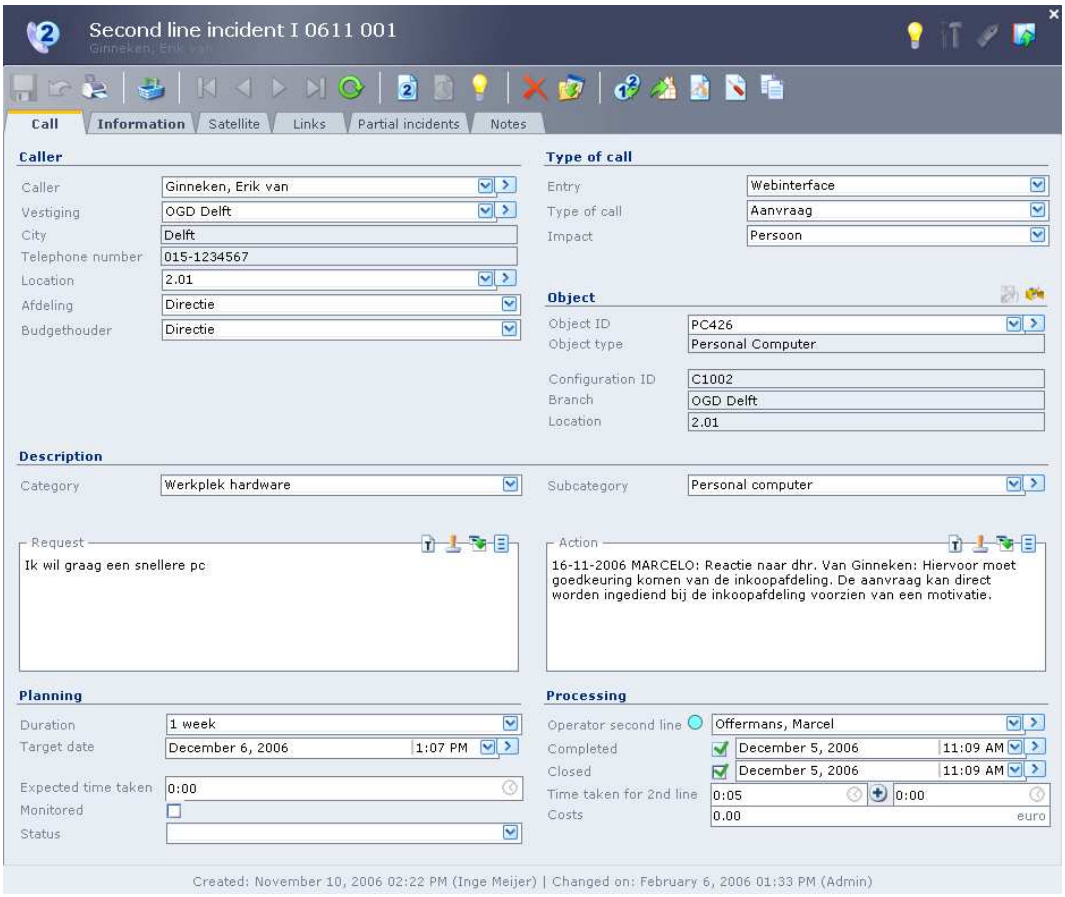


Figure 6.11: A TOPdesk incident

The script generated by a default Selenium IDE installation is listed in Table 6.2. As is clearly visible, Selenium uses strings generated by Mango for one-time use as fixed identifiers; while these strings remain unchanged when executing the script directly afterwards because of state information stored at the server, replaying it in a different session will fail due to unmatched identifiers. Also, some actions fail to execute, because Selenium does not wait for asynchronous communication initiated by Mango to complete. Apart from these failures, the use of these numeric identifiers make the script text itself difficult to understand by human readers, which is a disadvantage if one should want to debug or modify the script later. Manually modifying the script is a necessity in this case: some actions, like the addition of some text to a text area, is not recorded as intended, instead assuming a complete substitution of existing text by the value specified in the ”type” command. This can be seen in the action typing to target `c3_365`, where Selenium assumes the entire text area value should be re-entered, while in the recording session, a special button was pressed to add a timestamp to the text area, after which five characters were typed. All in all, the script recorded by Selenium is completely useless in a Mango environment.

| Command | Target | Value |
|---|---|---|
| open | /tas/secure/mango/window/3 | |
| click | c3_367 | |
| type | c3_365 | 09-10-2008 15:33 Admin: Done.\n 16-11-2006 MARCELO: Reactie naar dhr. Van Ginneken: Hiervoor moet goedkeuring komen van de inkoopafdeling. De aanvraag kan direct worden ingediend bij de inkoopafdeling voorzien van een motivatie. |
| click | c3_541 | |
| click | c3_538 | |
| type | c3_541 | 0:05 |
| click | c3_450 | |
| click | //div[@id='c3_1037']/div[2]/div[2] | |
| click | c3_27 | |

Table 6.2: Editing an incident in TOPdesk, recorded through Selenium IDE

As expected, controls present in the incident card as shown in Figure 6.11 were recognized and actions were recorded by Arsenic IDE as they were performed; the resulting script is listed in Table 6.3. Identifiers generated by Mango used sensible strings, representing values like underlying field names or button action names; these should be recognizable by test script maintainers. There were some exceptions, though: the first recorded command failing to capture the intended interaction was an attempt to add time to the incident's "time taken" field, as shown in Figure 6.12 and lines 9–12 of the script. This was caused by the fact that the field displayed to the right had no contextual information available for use by Mango, as it is only used to temporarily store a time value. Therefore, Arsenic IDE used the field's regular `id` attribute to identify it; this could be resolved by having the application developer define a fixed identification string for this field, as well as for the addition button next to it.

Unlike Selenium IDE, Arsenic IDE suggests assertions while a script is recorded. In this case, assertions regarding disabled buttons and modified values were suggested, as shown in Table 6.4: while editing the incident, its save and undo buttons were enabled, and some text box values were changed. After saving, these buttons were disabled again, as is reflected in the suggestion list. With regard to control identification, these suggestions obviously suffer from the same issues as other recorded commands, which accounts for the `value=0|textbox` identifier on line 4. A drawback to the DOM monitoring approach was also discovered through this case: the change of value in the "time taken" field by use of the addition button was not detected by Arsenic IDE, because its "value" attribute did not change; instead, the text box value was changed through code by altering its "value"

property. Because no browser event or JavaScript construct exists to monitor object property changes, this issue is nearly impossible to solve. Executing the script otherwise worked flawlessly; Arsenic waited for asynchronous communication to complete before processing the next command and generated identifiers were reusable in a new session.


Figure 6.12: An incident's "time taken" field


Figure 6.13: Part of a dialog showing "default text" available for insertion

| Command | Target | Value |
|---|---|---|
| open | /tas/secure/mango/incident ?unid=3u0002 | |
| waitForMangoPageLoaded | | |
| mangoTasMemoFieldactiondatetime-userstamp_extendedClick | mangohandle=actie | |
| mangoTasMemoTextareaKeyPress | mangohandle=actie | D |
| mangoTasMemoTextareaKeyPress | mangohandle=actie | o |
| mangoTasMemoTextareaKeyPress | mangohandle=actie | n |
| mangoTasMemoTextareaKeyPress | mangohandle=actie | e |
| mangoTasMemoTextareaKeyPress | mangohandle=actie | . |
| mangoRawTextBoxClick | mangohandle=value=0\|textbox | |
| mangoRawTextBoxKeyPress | mangohandle=value=0\|textbox | \8 |
| mangoRawTextBoxKeyPress | mangohandle=value=0\|textbox | 5 |
| mangoImageButtonClick | mangoid=538 | |
| mangoComboBoxDropdownbuttonClick | mangohandle=afhandelingstatusid | |
| mangoComboBoxItemClick | | Bijwerken in CMDB |
| mangoImageButtonClick | mangohandle=menu_action_save | |

Table 6.3: Editing an incident in TOPdesk, recorded through Arsenic IDE

| Command | Target | Value |
|---|---|---|
| assertAttribute | mangohandle=menu_action_undo@disabled | true |
| assertAttribute | mangohandle=menu_action_save@disabled | true |
| assertAttributeNotPresent | mangohandle=menu_action_archiveren@disabled | |
| assertValue | mangohandle=value=0\|textbox | 0:05 |
| assertValue | mangohandle=actie\|textarea | * |
| | * 09-10-2008 15:33 Admin: Done.<br /> 16-11-2006 MARCELO: Reactie naar dhr. Van Ginneken: Hiervoor moet goedkeuring komen van de inkoopafdeling. De aanvraag kan direct worden ingediend bij de inkoopafdeling voorzien van een motivatie. | |
| assertAttributeNotPresent | mangohandle=menu_action_undo@disabled | |
| assertAttribute | mangohandle=menu_action_archiveren@disabled | true |
| assertAttributeNotPresent | mangohandle=menu_action_save@disabled | |

Table 6.4: Assertions suggested by Arsenic IDE while editing an incident in TOPdesk

# Chapter 7

# Generalization

Now that a suitable robust testing solution has been found for Mango interfaces, the next challenge is to verify the practicality of this approach in combination with other toolkits. As mentioned before, two well-known kits have been selected for this purpose: Google Web Toolkit and NextApp's Echo, chosen because of their reputation and differences from both Mango and each other.

Due to the fact that these toolkits are not developed in-house as Mango is, an additional constraint has been imposed on toolkit code modification for testing purposes: all code should be usable in combination with a toolkit binaries released by the vendor, which means altering toolkit source itself is not allowed. The advantage of this approach is a decoupling of toolkit releases and test support code releases: as these are maintained by different developers and unlikely to be merged, it would be impractical to redo all changes to existing toolkit code with each toolkit version released.

## 7.1 Preparation

The first step towards a generic component-based Web interface testing tool is comprised of the externalization of Mango-specific functions in Arsenic to a separate section. Most of this functionality is covered by Arsenic's control mapping structure, as explained in section 6.2.2. Defining control interaction through this mapping syntax is expected to cover a major part of controls found in interface toolkits.

Toolkit-specific code will be needed in particular for defining script command termination conditions and more advanced scripted behavior. Due to different implementations of client-server communication, it is impossible to define a single termination condition useful for each toolkit. With regard to scripted behavior, implementation of functionality such as drag-and-drop is also done in a different way in each interface toolkit, making it necessary to define different functions to detect and execute such behavior.

On the server side, most effort will be needed to find a suitable method of identifier insertion, minimizing interference with regular interface code generation, while presenting identification data to Selenium in a uniform way. As this aspect has already been nontrivial to implement with Mango, even though toolkit code could be modified, extending GWT

and Echo to increase test-friendliness will be even more difficult.

## 7.2 Google Web Toolkit

The Web toolkit developed by Google shares some goals with Mango: both intend to eliminate the effort of having to develop a user interface with browser oddities in mind, by providing a Java API to their interface toolkit, which takes care of rendering the interface to a Web client. This is where similarities end; the implementation of GWT employs a different philosophy regarding this abstraction of Web interfaces. To begin with, Mango tends to be more of a platform-independent application framework than a Web framework, providing few to no Web-specific features in its API; GWT, on the other hand, has been developed specifically for Web interface purposes and contains features allowing a developer to finetune the interface by inserting hand-written JavaScript or HTML. This difference also echoes in the execution method of a GWT interface, because unlike Mango's approach to keep interface logic on the server side, GWT compiles interface code from Java to JavaScript, therefore executing without need for a Web server with Java support. An RPC interface is available to be able to communicate with other layers of the application, instead of allowing the developer to transparently call other Java code from the interface code, as is the case with Mango. In general, GWT can be considered much ”flatter” than Mango in terms of user interface aspects implemented. This makes an application developer using GWT much more aware of the fact that he is working in a Web environment, which can be considered both an advantage and a disadvantage.

### 7.2.1 Identifier generation

Regardless of the practicality of each approach for an application developer, GWT's flatness is a drawback to the current implementation of Arsenic. As the generation of identifiers makes heavy use of the models in Mango, a different method of identifier generation for any toolkit lacking such a concept is needed. Being more Web-based, GWT allows developers to define standard HTML attributes such as `id`, `name`, and `title`; these attributes, when manually specified, make some sense to use as identifiers in a testing environment, at least more so than generated id's, which are generated sparsely by GWT. Another useful feature of GWT is the inclusion of a so-called ”debug id” attribute, which has been introduced specifically as a convenience reference for testing purposes. This debug id implements a concept similar to the manual identifier override for Mango, making it a valuable resource for identifier generation. A drawback to all of these identifiers is the fact that they need to be specified manually by the application developer, as opposed to the Mango implementation, which is able to retrieve information from an included interface logic layer. An application based on GWT will obviously also include interface logic; however, this logic does not conform to a fixed interface, and it will need additional inspection code to have it yield identifier data.

### 7.2.2 Identifier insertion

Inserting these identifiers also is a far less trivial task with a precompiled toolkit. First of all, options are limited with regard to determination of the point in code execution where the identifier should be added to the element. While the Mango implementation allowed for changes to existing toolkit code, making it possible to include a hook at any desired location, the GWT implementation is dependent on the available Java API. Unfortunately, the only useful feature with this regard is a built-in event handler triggered when a page is loaded; there is no event handler available covering incremental interface updates, the aspect around which AJAX-based user interfaces revolve. GWT's Web-based nature, and one feature in particular, helps out: JSNI — short for JavaScript Native Interface — allows for the insertion of JavaScript code into Java classes written as part of a GWT interface. This enables a developer to tap directly into the DOM implementation of the browser executing the application. Although this counters GWT's concept of keeping browser specifics hidden, it provides the only access method to events not exposed by the Java API.

Features exploited by this method are indeed browser specific; Web browsers based on the Gecko engine implement the very useful `DOMNodeInserted` event to watch for DOM changes, while Internet Explorer has no equivalent. Therefore, in tool implementation for IE, one will need to resort to more cumbersome and less accurate methods to hook into DOM modification events. One of these methods consists of overriding the `appendChild` procedure available in each DOM element. While using this procedure is by far no way to be sure every element addition is detected in most Web applications, GWT's implementation relies on `appendChild` to do all DOM insertions; if there was no such guarantee, one would probably need to implement a polling mechanism to do DOM update checking. Due to the interaction possible through JSNI, these JavaScript event handlers and method overrides are capable of doing Java method callbacks — which are in fact translated to JavaScript methods when compiled — taking care of the actual identifier generation and insertion.

One aspect of the semi-automatic annotation of an application for testing purposes not yet treated, is the method chosen to be used by the application developer to include test-specific code in the GWT application. Fortunately, Google Web Toolkit provides a modular structure for application loading, using an XML configuration file where modules to be loaded should be specified. This means that all code involved in generation and addition of identification data can be stored within one module, keeping the application and GWT itself free of modification apart from module inclusion within the configuration file. An application developer will then only need to add this module to the configuration file of the application under test, as shown in Figure 7.1.

## 7.3 Echo

The Echo Web Application Framework — currently stable at version 2, thus commonly known as Echo2 — resembles Mango more closely than GWT with respect to architectural layout. User interfaces are implemented in Java, and at runtime this Java code is kept on the server side, relying on AJAX-based communication to build and update the interface DOM at the client. Like Mango, Echo has been designed as a platform-independent interface

```
<!-- Enable debug ID. -->
<inherits name="com.google.gwt.user.Debug"/>
<set-property name="gwt.enableDebugId" value="true"/>

<!-- Add test attributes -->
<inherits name='com.topdesk.test.GwtTest'/>
```

Figure 7.1: Tags added to a GWT module configuration to support Arsenic testing

toolkit, rather than a pure Web interface toolkit; unlike Mango, however, Echo does not provide an extensive layer of models to back interface components with logic. This feature is also missing from GWT, which makes it more difficult to define sensible defaults for test identifiers generated by the testing layer.

### 7.3.1 Identifier insertion

Due to the inherent openness of JavaScript and the JavaScript-focused implementation of GWT, this toolkit eventually provided many entry points for insertion of these test identifiers. Echo, due to its Java-based nature, is mostly closed to external hooks. However, it does allow custom components to be registered with its rendering engine. This leads to the idea of creating a component which does not have any visual elements, but instead adds test attributes to each of the interface's other components through its Web rendering counterpart. An obvious drawback to this approach is the fact that one would need to be able to add an instance of this component to the application under test; an option not necessarily available in every Echo interface, since components are allowed to reject child components, possibly leading to a situation where the test component does not fit anywhere in the interface hierarchy. Also, the architectural decision of having a single component influence all other components is not a very elegant solution with regard to separation of responsibility.

Although creation of a dummy component is not the solution we are looking for, manipulating an aspect of the application itself seems necessary when dealing with a toolkit closed to modification. Another aspect of the Echo implementation is useful here: deployment of an application is done through regular Java servlets, which makes it possible to configure the Java server to send application requests to a test servlet, effectively proxying the application servlet. An added advantage of Echo in this approach is the fact that the servlet is only used to obtain an instance of a separate application class, which takes care of further application behavior; this means proxy behavior implementation is very limited. Through this functionality, a reference to the application instance is available; this reference can then be used to add identifier insertion hooks to the application.

This is where Echo's platform independence interferes: although user interface state is preserved by the framework, the state of the DOM at the client side is not preserved at the server side when performing such actions as refreshing the page. Furthermore, inter-

face redrawing events are not triggered when a client refreshes a page, which results in inserted component identifiers being lost after a page reload. Because of the application being unaware of Web-related issues, a move back to the servlet level is required. Fortunately, another injection point is available at this layer: Echo uses a so-called ”service registry” to be able to process certain types of requests done by its Web layer. It is possible to manipulate this registry in order to replace a specific service with a custom implementation. Doing this allows for insertion of test attributes with each request done from the client side, which is necessary to capture complete DOM refreshes.

### 7.3.2 Identifier generation

Having found a suitable solution for identifier insertion, we once again turn to the issue of identifier value generation. While Mango provides a wide array of models to define logic or data bindings for its components, Echo, like GWT, focuses on pure interface definition, using the common notion of event handlers to leave interface logic implementation to the application developer. However, also like GWT and unlike Mango built-in features, Echo components may be assigned a static identifier string, which is usable for generation of a test identifier. The fact remains that application developers will stay responsible for explicitly defining component identifiers, as opposed to implicit definition by use of models in Mango.

An issue relevant to Mango as well as Echo is timing of execution of test script commands, due to asynchronous updates done by both toolkits. Both toolkits include an internal manager for outstanding `XMLHttpRequest`s, allowing Selenium code to monitor this manager for activity and throttle script execution accordingly.

# Chapter 8

# Conclusions and Future Work

## 8.1 Contributions

The main contribution of this project is the implementation of a testing tool aimed at approaching a Web application's user interface as a coherent collection of controls, rather than a DOM tree of meaningless elements. This tool is meant to be used with applications built through toolkits providing controls often seen in application interfaces as their building blocks; more specifically, several Web interface toolkits currently available have been extended to provide contextual information to the client. This information is then used by a modified version of Selenium, named Arsenic, to capture user interaction in test scripts, which can be executed afterwards. Test script syntax has focused on both reliability of reproduction, meaning scripts will not break with every minor modification to interface code, and maintainability by software testers. Arsenic's differences compared to Selenium are related mainly to the IDE's recording phase, which has been modified to be able to cope with interaction methods used regularly in AJAX interfaces: while Selenium IDE was designed to record clicks on regular links and buttons and filling out forms by setting input element values, Arsenic IDE recognizes and records any type of interaction with the interface defined by the toolkit developer. Definition of these interactions is a fairly quick and trivial process which can be carried out by anyone with some knowledge of said toolkit's internals.

Tool implementation has focused mainly on Mango support. Due to this, Arsenic's maturity level in combination with Mango-based interfaces is fairly high; however, because Arsenic relies on modifications to Mango not yet included in its main branch, and because Mango itself has not yet been released to the public, Arsenic will remain an internal tool used at TOPdesk until Mango's release. Due to OpenQA's projects, such as Selenium, using the Apache 2.0 license, redistribution of modified source code will be possible at any time. Arsenic's GWT and Echo extensions have been implemented as a proof of concept, and have not entered a phase where they are suitable for release to the general public.

## 8.2 Conclusions

One thing that has become apparent from earlier research into related work is the fact that one can go a lot of ways with automated dynamic functional testing of AJAX interfaces. All of them are obviously built on the principle of interacting with the interface and asserting the correctness of the results of these interactions. Some of these use models based on mathematics or software engineering, but one in particular stands out due to its simplicity: record and playback. While knowledge of the principles behind these models is required to be able to operate a tool based on such a model, record and playback is an intuitive, practical and therefore easy to use approach. For this reason, the fact that its playback engine can be fitted into a framework based on any of these other methods, and its popularity with software engineers, this approach is the way to go for a first generation AJAX testing tool.

More technical issues have also been discussed in the aforementioned research, the first one regarding the interaction method with the Web application UI. As noted in the discussion about portability, testing for browser specific quirks is of great importance, nullifying the value of the browser emulator, which essentially forms an entirely new browser not at all interesting to test for. All other options discussed interact with real-world browsers; both the plugin and embedder require knowledge and availability of a browser API capable of building a playback tool upon. The proxy approach allows for keeping framework code completely separate from the browser, instead intercepting the browser's connections and communicating with the Web application through these hijacked lines. While this is a suitable solution when one needs to test for a browser difficult to extend or embed, its architecture, usually designed around JavaScript injection, is somewhat questionable: one would need a stand-alone testing framework for managing and executing scripts, along with proxy code injecting JavaScript into the Web application, and browser-dependent JavaScript code to control the interface.

This has lead to the question about whether to extend an existing Web 1.0 tool or to implement a completely new one. Because of the fact that generated user interfaces are still made up of HTML, test tool development was expected to benefit from extending an existing HTML testing tool, and some of the tools evaluated were quite promising with regard to their extendibility for AJAX. Especially Selenium, which supports many browser platforms and has already proven itself extendible through the implementation of GITAK, appeared to be a suitable candidate for extension. In this project, Selenium was therefore chosen as an extension platform, which also set the choice of method with regard to browser interaction, as all of Selenium's plugin code is readily available. It also provided much of the code needed to support DOM-level testing; the main focus of the AJAX tool could therefore be shifted to build upon this layer. Because these HTML-based interfaces are comprised primarily of machine-generated code difficult to consistently identify by any testing tool, a method to improve on this issue needed to be defined. It was established that adding a string describing its context to each control would be beneficial to both the tool and its user, as it also improves readability of test scripts.

Because of company interest, the primary target toolkit to develop for was Mango. Focusing on this kit, Arsenic was developed, taking into account the fact that a phase of generalization to other toolkits would be executed later on. Due to close collaboration with

TOPdesk's quality assurance department, the resulting tool is both easy and practical in use, recognizing and interacting with Mango code in a way virtually seamless to the user.

The next step, generalizing the Mango tool implementation to other toolkits, proved to cost less effort than the initial extension development phase, as was expected. Assumptions on similarities in toolkit architecture turned out to be mostly true; a notable example of an advantage of Mango's architecture with this respect is its use of models as a standardized method of defining interface logic. The lack of such a standard in other toolkits hindered suitable automated identifier selection, preventing the goal of minimal effort by application developers in the testing process from being achieved. This means developers still need to explicitly define suitable identifiers for controls in GWT and Echo, apart from cases where interface text can be used as an identifier — which is not much of an improvement from regular HTML testing tool behavior.

The extent to which generalization was possible turned out to be fairly high. Most of the differences between toolkits stem from diverse methods of client-server interaction and synchronization, instead of alternative control implementations. As a result, a large portion of Mango testing tool code could be used for GWT and Echo tool development, supplemented by a small segment of code specifically dealing with toolkit quirks and custom interaction methods.

## 8.3 Discussion/Reflection

Selecting Selenium as a platform for extension turned out to be a good decision. Its extension interface provided sufficient functionality to build the desired tool, while the openness of the remainder of its code allowed for the implementation of other missing features. More time was needed to modify interface toolkits to cooperate with Selenium than to modify Selenium to cooperate with interface toolkits; one might argue that this is a matter of division of responsibilities between tool and toolkit, although moving a larger slice of responsibility to Arsenic would require it to have knowledge of toolkit internals to an undesireable level. Implementation results were satisfactory: the short case study presented in section 6.4 has shown that reliable script recording and execution using Arsenic IDE can be done by software testers with some knowledge of interface controls and events.

One should keep in mind that this project does not intend to favor capture-and-replay over other functional regression testing techniques. The main concept introduced has been a shift of paradigm from defining test scripts in terms of HTML elements to usage of virtual controls, as they are defined by interface toolkits; this concept could be used in any testing technique that needs to store references to individual interface elements. The decision to base tool implementation upon capture-and-replay has been made because of the tool's target audience: software testers without a degree in computer science, who will have difficulty to comprehend the complexity of testing based on concepts such as finite state machines. Also, the particular level of involvement of individual controls within capture-and-replay has made it a good choice to use as an implementation platform; modification of its recording phase has been a nice opportunity to implement control detection and interaction. The actual modifications were influenced by limitations in Selenium's scripting

syntax, as discussed in section 6.2.1. Changing the command-target-value syntax itself was not an option, because all of Selenium's components rely on it for test script parsing and execution.

Some of the goals set early on were not achieved. One of these goals was to directly monitor all events dynamically attached to DOM elements, determining whether an event was worth recording as it was triggered, instead of picking a set of events considered useful for particular toolkit controls in advance. Failure of this goal has been caused mainly by the fact that Selenium does not support this level of dynamic action definition. It is assumed though, that other testing tools suffer from this same issue, and that it is a difficult feature to improve upon.

Also, in contrast with time needed to modify Selenium, toolkit modification with the intent of improving automated testing facilities proved to be more time-consuming than expected. Neither GWT nor Echo are designed to allow a developer to modify generated HTML, so backdoor methods were needed in order to successfully change code generation. An opportunity for improvement therefore lies with interface toolkit developers: making generated code more suitable for robust automated testing.

## 8.4 Future work

An architectural point of improvement for Arsenic itself will allow the tool to interact with the application server in order to discover toolkit capabilities, such as available control types, control interaction methods, and synchronization status. This improvement would save the effort needed to manually define these items; also, it would keep toolkit-specific code with the toolkit itself, while reverting Arsenic to a toolkit-independent interface testing suite.

Concepts introduced in this project may also benefit other research related to robust automated AJAX-based interface testing. In particular, most of this research focuses on automated traversal of interface states, defining methods to automatically find interaction paths to any state deemed relevant. However, these interaction paths reference interface components by the very same identifiers that change with each minor interface code modification, such as the `id` attribute and XPath expressions; using identification methods described in this report would yield much more robust state transition graphs.

# Bibliography


[1] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with fsms. *Software and Systems Modeling*, 4:326–345, 2005.

[2] Fevzi Belli. Finite-state testing and analysis of graphical user interfaces. In *ISSRE '01: Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01)*, page 34, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[3] Justin Boekestijn. Testing advanced web interfaces, research report. 2008.

[4] Bert Bos. Setting the scope for light-weight web-based applications. *World Wide Web Consortium*, 2004.

[5] S.H. Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification & Reliability*, 11(2):97–111, 2001.

[6] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving web application testing with user session data. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 49–59, Washington, DC, USA, 2003. IEEE Computer Society.

[7] F. Fraikin and T. Leonhardt. From requirements to analysis with capture and replay tools. *PI-R 1/02, Software Engineering Group, Department of Computer Science, Darmstadt University of Technology*, 2002.

[8] H.W. Gellersen and M. Gaedke. Object-oriented web application development. *Internet Computing, IEEE*, 3(1):60–68, 1999.

[9] Minmin Han and Christine Hofmeister. Modeling and verification of adaptive navigation in web applications. In *ICWE '06: Proceedings of the 6th international conference on Web engineering*, pages 329–336, New York, NY, USA, 2006. ACM.

[10] Andreas Lecerof and Fabio Paterno. Automatic support for usability evaluation. *IEEE Transactions on Software Engineering*, 24(10):863–888, 1998.

[11] N. Mansour and M. Houri. Testing web applications. *Information and Software Technology*, 48(1):31–42, 2006.

[12] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 260, Washington, DC, USA, 2003. IEEE Computer Society.

[13] Atif Memon, Martha Pollack, and Mary Lou Soffa. Plan generation for gui testing. *Proceedings of The Fifth International Conference on Artificial Intelligence Planning and Scheduling*, pages 226–235, 2000.

[14] Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling ajax by inferring user interface state changes. In *Proceedings of the 8th International Conference on Web Engineering (ICWE'08)*, pages 122–134. IEEE Computer Society, July 2008.

[15] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. IEEE Computer Society, 2009.

[16] Linda Dailey Paulson. Building rich web applications with ajax. *Computer*, 38(10):14–17, 2005.

[17] Dave Raggett. *Raggett on HTML 4*. Addison-Wesley, Harlow, England; Reading, MA, USA, 1998.

[18] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[19] D. Richardson, O. O'Malley, and C. Tittle. Approaches to specification-based testing. In *TAV3: Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, pages 86–96, New York, NY, USA, 1989. ACM.

[20] Sreedevi Sampath, Valentin Mihaylov, Amie Souter, and Lori Pollock. A scalable approach to user-session based testing of web applications through concept analysis. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 132–141, Washington, DC, USA, 2004. IEEE Computer Society.

[21] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. Automated replay and failure detection for web applications. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 253–262, New York, NY, USA, 2005. ACM.

[22] Marlon Vieira, Johanne Leduc, Bill Hasling, Rajesh Subramanyan, and Juergen Kazmeier. Automation of gui testing using a model-driven approach. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 9–14, New York, NY, USA, 2006. ACM.

[23] Filippos I. Vokolos and Elaine J. Weyuker. Performance testing of software systems. In *WOSP '98: Proceedings of the 1st international workshop on Software and performance*, pages 80–87, New York, NY, USA, 1998. ACM.

[24] A.M. Wichansky. Usability testing in 2000 and beyond. *Ergonomics*, 43(7):998–1006, 2000.

# Appendix A

# Getting started with Arsenic IDE

*This guide has been released to TOPdesk quality assurance employees during test runs of the Selenium IDE extension, Arsenic IDE.*

Arsenic IDE is a tool created to allow people involved with developement of a Mango application to set up automated user interface tests through the concept of ”record and playback”. Arsenic is derived from Selenium[1], a general-purpose Web user interface testing suite. While in theory Selenium is also supposed to be able to deal with Mango interfaces, experience has shown that these test scripts are unmaintainable; therefore, Arsenic was conceived to focus on testing Mango interfaces in a maintainable manner. This guide is meant to familiarize you with some basic principles of automated user interface testing, and to explain features of Arsenic different from Selenium IDE.

## A.1 Automated user interface testing

If you are reading this, it is assumed that you are already somewhat familiar with software testing in general. This document will be limited to a short explanation of the benefits of automated user interface testing and the concept of ”record and playback”, also known as ”capture and replay”. Automation of user interface tests, when applied in a sensible way, saves human testers a lot of work related to clicking through the same interfaces over and over again. Of course, automation is only useful when you know the interface to be tested in advance: to set up a test scenario, you need to know how you can interact with the interface, and how it will respond to your actions. Therefore, automated user interface testing is mostly used for so-called ”regression testing”, to check whether an existing interface still works as expected after some changes to it or related code have been made. A popular method of interface testing is known as record-and-playback and is usually executed as follows: first, a tool is used to record a script consisting of actions executed by the user; then, some verifications are added to the script by the user of the tool, checking whether the interface responds to the user’s actions as expected. Finally, the script is added to the collection of test scripts already recorded, to be executed automatically on a periodic basis.

[1] `http://selenium.openqa.org`

## A.2 Installation

Because Arsenic IDE, like Selenium IDE, has been implemented as an extension for Mozilla Firefox, installation is fairly straightforward. Drag and drop the XPI file to any open Firefox window to start installation; Firefox will guide you through the rest of the process.

## A.3 Launching the IDE

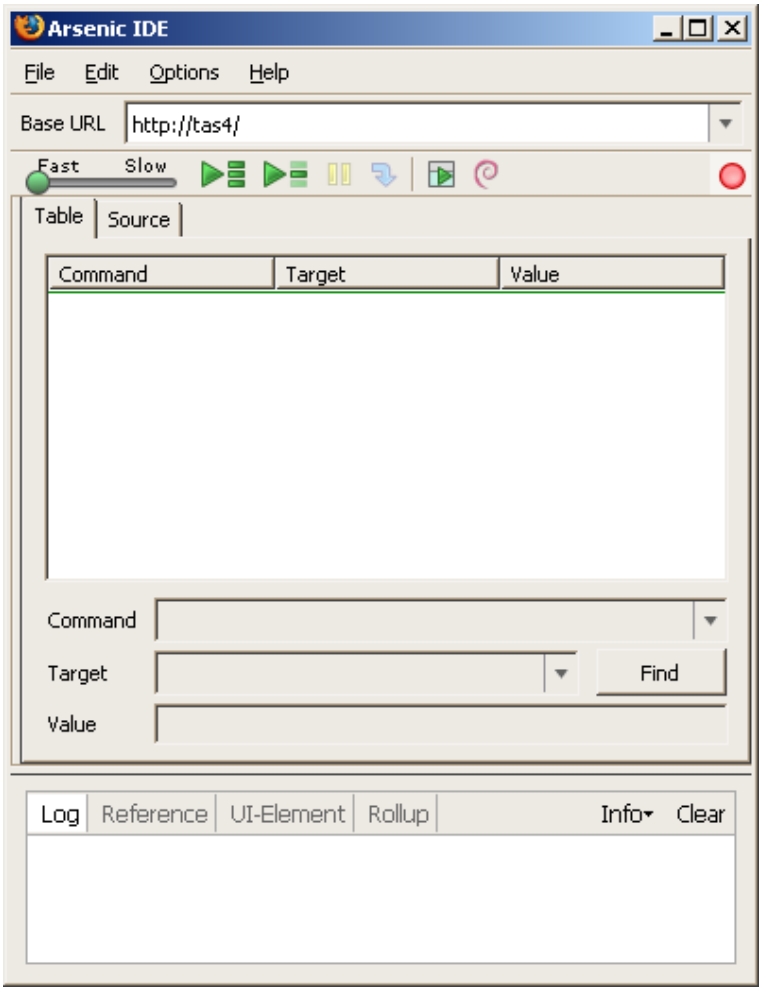


Figure A.1: The initial Arsenic IDE interface

After installation, a new item is available under Firefox' *Tools* menu, appropriately named *Arsenic IDE*. Clicking this item will launch the Arsenic's recording interface and immediately switch to recording mode. From this point, every action executed within a Web page will be recorded by Arsenic.

## A.4 Recording a script

Before you start recording a script, I need to stress once more that Arsenic has been designed to test Mango interfaces. Assuming you are using Arsenic to test a section of the TOPdesk user interface, you have to realize its conversion to Mango is not yet complete, and correct behavior of the tool is not guaranteed when navigating to non-Mango sections of TOPdesk. To minimize the odds of this behavior occurring, make sure the Mango interface you want to test has been opened outside TOPdesk's frame set before starting Arsenic IDE.

In this example, we are going to demonstrate the recording of creation of a new person in TOPdesk. The easiest way to open a new person card outside TOPdesk's frame set, is to navigate to the *Supporting files* bumper page, right-click the link to *Person* under the *New* header, and use Firefox' *Open Link in New Tab* option. You should now be presented with an empty person card in a separate Firefox tab. Next, launch Arsenic IDE by selecting *Arsenic IDE* from the *Tools* menu; you are set to record a test script!

Play around with the card for a while and notice Arsenic recording you actions; for example, after entering a surname, first initials, and a branch, your script may look like Table A.1.

Notice that Arsenic IDE records every key you press as a different action; this has been done to ensure Mango treats everything done by Arsenic as though a real user were filling out the person card, which is especially important for fields with autocompletion features.

Continue by entering values into all mandatory fields on the *General* tab, then clicking on the *Private* tab to fill in one more field. Enter a *Date of birth* by expanding the calendar control and navigating to a date of choice, then save the card by clicking the corresponding icon. Doing this should add lines similar to Table A.2 to your script.

This completes the recording phase of this script. Toggle the record button in the upper right corner of Arsenic IDE to disable recording mode.

## A.5 Adding verifications

The next step in creating a test script is result verification, to make sure actions executed earlier have resulted in the creation of a person with the details supplied. This step is nearly as easy as script recording: it mainly requires clicking the interface and Arsenic IDE itself. First of all, we will check whether the date of birth entered through the calendar control matches the date expected. To do this, right-click the input box containing the date of birth; a context menu should pop up, having a few more items than you would expect. Click the item starting with *verifyValue*, a line verifying the date of birth is now added to the script. Next, we are going to check whether the value typed into the branch field matches the expected result. Turn on recording mode again, go back to the *General* tab, turn off recording mode, and repeat the *verifyValue* step for the branch field. Finally, let us verify whether Mango has updated the page title to show the person's name and department. Find a spot on the page where Mango allows display of a context menu (text boxes are usually a good candidate) and right-click, then click the entry starting with *assertTitle*. The lines added to your script should look something like Table A.3.

| Command | Target | Value |
|---|---|---|
| open | /tas/secure/mango/person?action=new&status=1&action=new&status=1 | |
| waitForMangoPageLoaded | | |
| mangoTextBoxClick | mangohandle=achternaam | |
| mangoTextBoxKeyPress | mangohandle=achternaam | B |
| mangoTextBoxKeyPress | mangohandle=achternaam | o |
| mangoTextBoxKeyPress | mangohandle=achternaam | e |
| mangoTextBoxKeyPress | mangohandle=achternaam | k |
| mangoTextBoxKeyPress | mangohandle=achternaam | e |
| mangoTextBoxKeyPress | mangohandle=achternaam | s |
| mangoTextBoxKeyPress | mangohandle=achternaam | t |
| mangoTextBoxKeyPress | mangohandle=achternaam | i |
| mangoTextBoxKeyPress | mangohandle=achternaam | j |
| mangoTextBoxKeyPress | mangohandle=achternaam | n |
| mangoTextBoxClick | mangohandle=voorletters | |
| mangoTextBoxKeyPress | mangohandle=voorletters | J |
| mangoTextBoxKeyPress | mangohandle=voorletters | . |
| mangoComboBoxTextboxClick | mangohandle=vestigingid | |
| mangoComboBoxTextboxKeyPress | mangohandle=vestigingid | t |
| mangoComboBoxTextboxKeyPress | mangohandle=vestigingid | o |
| mangoComboBoxTextboxKeyPress | mangohandle=vestigingid | p |
| mangoComboBoxTextboxKeyPress | mangohandle=vestigingid | d |
| mangoComboBoxTextboxKeyPress | mangohandle=vestigingid | e |
| mangoComboBoxTextboxKeyPress | mangohandle=vestigingid | s |
| mangoComboBoxTextboxKeyPress | mangohandle=vestigingid | k |
| mangoComboBoxTextboxKeyPress | mangohandle=vestigingid | \32 |
| mangoComboBoxTextboxKeyPress | mangohandle=vestigingid | n |

Table A.1: Script state after initial recording

Congratulations, you have written a complete Arsenic test script! You may choose to save the script by clicking *File, Save Test Case* from Arsenic IDE's menu. Next, we will check whether the script executes as expected.

## A.6 Executing the script

To execute the script we have just created, simply press the play button near the top of the Arsenic IDE screen — the toolbar contains two play buttons, either one is fine if you have only one script loaded. Arsenic will now try to reproduce your actions, and check whether your assertions were correct. Successful assertions will be shown with a green background, while failed assertions will turn red. If an action fails to execute, it will also turn red and

| Command | Target | Value |
|---|---|---|
| mangoTabButtonClick | mangohandle=tab_private_data | |
| mangoButtonClick | mangohandle=geboortedatum\|dropdownbutton | |
| mangoImageButtonClick | mangohandle=datepicker\|previousmonth | |
| mangoLabelClick | mangohandle=datepickerlabel2008.7\|20080826 | |
| mangoButtonClick | mangohandle=button_ok | |
| mangoImageButtonClick | mangohandle=BusinessAction::save | |

Table A.2: Script state after recording date of birth

| Command | Target | Value |
|---|---|---|
| verifyValue | mangohandlepath=/tab_private_data_page/ geboortedatum/geboortedatum | August 26, 2008 |
| mangoTabButtonClick | mangohandle=tab_general_data | |
| verifyValue | mangohandle=vestigingid\|textbox | TOPdesk Nederland |
| assertTitle | Boekestijn, Justin - Systemen | |

Table A.3: Script state after adding verifications

Arsenic will abort script execution. At this point, if the script operates as expected, you are finished — short of adding the script to the automated test set, a topic beyond the scope of this document. Otherwise, you may find Arsenic IDE's debugging and editing features useful in tracking down and solving the issue. Most of these features are available from a context menu: right-clicking a script action allows you to execute a single recorded action, set a breakpoint, or move the script's starting point; also, it is possible to reorder script actions by cutting, copying and pasting them. For some more pointers, refer to the common pitfalls section for analysis of some common problems.

## A.7 Common pitfalls

The script recording phase is the most error-prone part of script creation. Usually, a malfunctioning script is caused by the recorder not registering user actions as they are intended. Apart from Arsenic IDE's debugging features, the following pointers may prove helpful in solving script issues:

- Recording complex text manipulation is currently not possible: Arsenic will not record text selection, cutting, copying and pasting of text correctly. You can work around this by not doing any text selection while recording a script.
- Arsenic will always start manipulation of text boxes at the last character, clicking at a particular position in a text box will not be correctly recorded. However, using the arrow keys to move the cursor within a text box is recorded and executes accordingly.

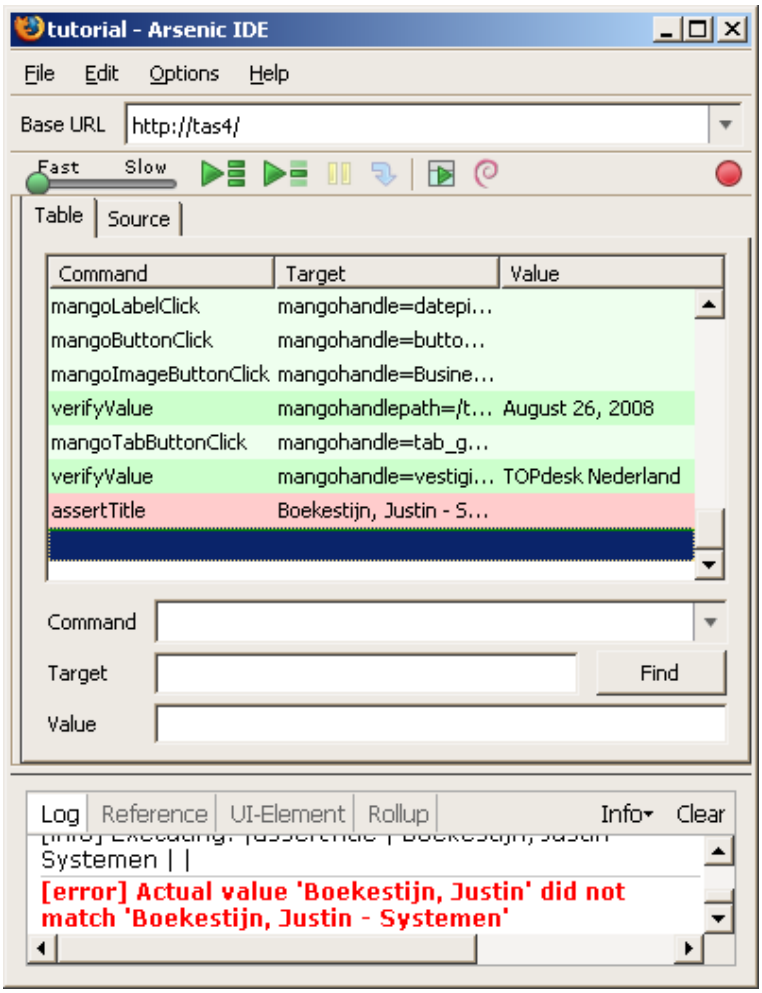


Figure A.2: Arsenic IDE, showing test results

- A related issue concerns incremental updates: if a test script opens the person card for Mr. Smith, adds two e's to his surname, verifies the surname to be "Smithee", and saves the card, the next execution of this script will change Mr. Smithee's surname to "Smitheeee" and fail verification. A workaround for this issue is to order the script to clear the input before appending text; to do this, use the "type" command combined with an empty value.

- Check whether all recorded actions consistently yield the same result. For instance, the calendar control used in the above example will always show the current month when opened. If Arsenic is ordered to "click the first day of the month shown", it will select a different date of birth if the script is executed a month later.

- Sometimes, actions recorded will use a command not starting with "mango", while the action clearly refers to a Mango control. This is most likely a bug in Arsenic and

should be reported to the author.

- Another recording issue pops up when you interact with ”weakly referenced” controls; usually, actions involving such a control will not have a target starting with ”mangohandle=”. If you encounter such a control, report it to Arsenic’s author.

## A.8 Further reading

Be sure to read Selenium documentation[2] for more detailed information on Selenium IDE.

[2] `http://selenium.openqa.org/documentation/`