Albatross: Seaside Web Applications Scenario Testing Framework

Andrea Brühlmann, abrue@students.unibe.ch
Supervised by: Adrian Lienhard
Software Composition Group
University of Bern, Switzerland

September 2006

Abstract

Seaside is a framework for developing sophisticated web applications in Smalltalk. One thing missing until now has been a way to automatically test the running applications in a web browser. We could open a browser and test some scenarios by hand. This is not very effective for larger applications and for regression testing though, so we need a way to write automatic tests for our web applications.

Albatross is the key to this problem, because it allows us to write tests directly in Smalltalk using the unit testing framework. It opens the web application in an external web browser and simulates user interactions. It provides access to the running and rendered web application and at the same time to the model of your application.

There is even no need of bothering with HTML tags and ids, because Albatross has cleverer ways to find out what to do. It finds form fields just by identifying the corresponding label text or clicks on links just by locating the displayed link text.

Contents

1 Introduction **2**
- 1.1 Seaside Web Application Framework . . . 2
- 1.2 The Problem of Testing Seaside Applications . . . 2
- 1.3 Albatross: SUnit Tests in an External Web Browser . . . 3
- 1.4 Related Work . . . 3

2 Example: Pier Test **5**
- 2.1 Edit Page . . . 5
- 2.2 Make Sushi Shop . . . 6
- 2.3 Buy Akami Maguro . . . 8

3 Implementation **10**
- 3.1 Environment . . . 10
- 3.2 Server - Client Communication . . . 10
  - 3.2.1 Tasks and Results . . . 10
  - 3.2.2 Test Setup . . . 12
- 3.3 Albatross Test Messages . . . 12
  - 3.3.1 Actions . . . 12
  - 3.3.2 Tests . . . 13
  - 3.3.3 Finding Elements . . . 14
  - 3.3.4 Model Access . . . 15
- 3.4 Albatross JavaScript Functionality . . . 15
  - 3.4.1 Prototype . . . 15
  - 3.4.2 Albatross js . . . 15

4 Conclusion **18**

Chapter 1

Introduction

1.1 Seaside Web Application Framework

Seaside [2] is a framework for developing sophisticated web applications in Smalltalk [4]. It provides a layered set of abstractions over HTTP and HTML that lets you build highly interactive web applications. Seaside includes programmatic HTML generation, callback-based request handling, embedded components and modal session management. All this enables the programmer to develop object-oriented web applications instead of creating page by page and thinking about ids and HTML tags. It also supports Ajax (Asynchronous JavaScript and XML) [3], which allows handling HTTP requests within a page without reloading the whole page.

1.2 The Problem of Testing Seaside Applications

One thing missing until now has been a way to automatically test the applications on the user interface level. Tests have to be done manually by opening a web browser and running through some scenarios. This is not very effective for larger applications and for regression testing, so automatic tests are needed.

In Squeak [9], there is a unit testing framework called SUnit [1]. With this, an applications model can be tested very easily. There is another framework especially for testing Seaside applications: Seaside Testing Framework [7]. This tests the generated HTML pages and has access to the objects in the application, but does not run the application in an external web browser. The behaviour of a running application on the client side is impossible to test with the Seaside Testing Framework which is staying on the server side. Ajax and JavaScript cannot be tested this way, not to mention browser compatibility.

1.3 Albatross: SUnit Tests in an External Web Browser

What is needed is an interface between Squeak and an external web browser. It should be able to start a Seaside application in the browser and to communicate with the browser and an SUnit test case. In the browser, it should simulate user interactions or read values and return answers to the test case. The test case should give a protocol of actions and in between, make assertions on returned values.

Ideally, the test case should also have direct access to the instance of the Seaside application. With that, it could get information from the model or even change it if wanted.

In addition to this interface, there needs to be some functionality to allow easy access to the elements on the web page. It should be possible to tell the interface to click on a button with the text "Ok", for example. Or to find the form field that belongs to "Name:". Like this, test cases would be quickly written and the programmer would not have to deal with ids or HTML tags.

Albatross has been designed to meet these needs. It consists of a Squeak package and a JavaScript library. The JavaScript provides the above mentioned intelligent functionalities. The Squeak package handles the communication between the test case and the web browser and the transformation of test tasks to JavaScript code. Each transformed test task is sent to the browser, where it is executed and the result is sent back to the test case. This is shown in Figure 1.1.

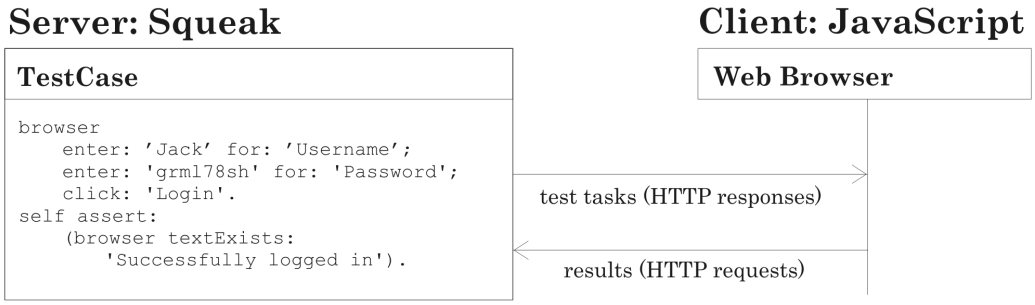


Figure 1.1: Interaction between test case and web browser

1.4 Related Work

- Seaside Testing Framework [7] tests the generated HTML pages and has access to the objects in the application. It supports no Ajax/JavaScript testing because it does not run the application in a browser but tests it from the server side.
- Squelenium [8] is an adaption of Selenium RC [5] for Squeak. Selenium RC is a test

tool that allows you to write automated web application UI tests in any programming language against any HTTP website using any mainstream JavaScript-enabled browser. It has been ported to Squeak to provide a possibility to write SUnit tests which are executed by Selenium. It does not provide access to the Seaside objects though and is a huge package with dependencies to an external Selenium library.

- General web application testing: there are several commercial products that provide functional testing. They have no access to the objects of the application and have no connection to SUnit tests, but they usually have a test runner in the browser with a rich graphical user interface.

Chapter 2

Example: Pier Test

To illustrate how Albatross works we introduce an example application. It shows how a user scenario can be implemented in Albatross, how assertions can be made and how the objects of the application can be accessed. As example application, we take Pier [6], a Seaside content management and Wiki system.

We want to test the following scenario:

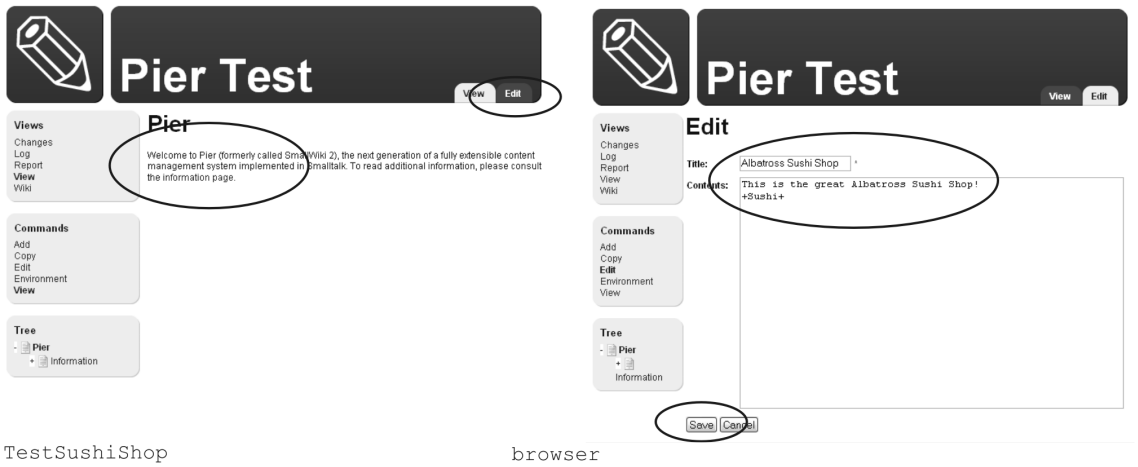
- We edit the root page and include a component named Sushi
- Then we click on that link to create the Sushi component and make it a Sushi store (this is an included Seaside demo component)
- Back on the root page, we browse the Sushi shop and add an Akami Maguro to the shopping cart

The rest of this chapter explains how the Pier test scenario can be implemented using Albatross.

2.1 Edit Page

To start, we create a class `PierTest` as a subclass of `ATestCase`. We write a method createTestComponent that returns a new instance of the Pier root component. For the test scenario, we add a method called testSushiShop. In Figure 2.1 the scenario start and the corresponding code lines are shown.

First, we want to check if everything is as expected. Then we want to click on the link with the text "Edit". This will open the editor for the root page. We enter some text for the title and the contents of the page, including "+Sushi+". This is the Pier syntax for including a component into the current page. To finish editing, we want to click the button with the text "Save" on it.




```
TestSushiShop
   self assert:
      (browser textExists:
         'Welcome to Pier').
   browser click: 'Edit'.
```

```
browser
   enter: 'Albatross Sushi Shop' for: 'Title';
   enter: 'This is the great Albatross Sushi Shop!
   +Sushi+' for: 'Contents';
   click: 'Save'.
```

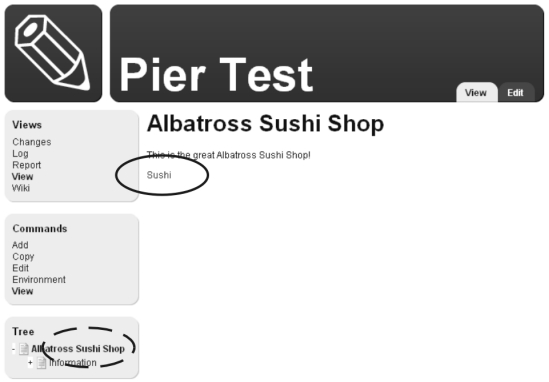
Figure 2.1: Pier example scenario: editing a page

2.2 Make Sushi Shop

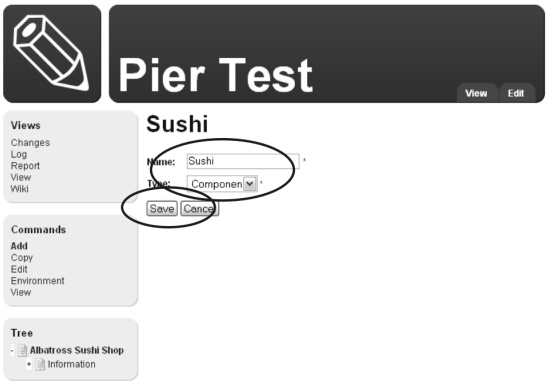
Now we want to create the Sushi shop as shown in Figure 2.2. To do this, we have to click on the created word 'Sushi'. The problem with that is that there is another link on the page which contains 'Sushi' - the title of our page in the navigation tree (See Figure 2.2 top left). So we specify our Sushi-link by saying that it is in the contents area: we give a CSS-selector that takes all elements that are in an element with the class 'contents'.

As for text input fields, we can enter values for drop down menus. By writing `enter: 'Component' for:'Type'`, we select 'Component' from the select box (Figure 2.2 top right). We save this with a click on 'Save'. For the component class, we select 'Store' from the select box and save that.

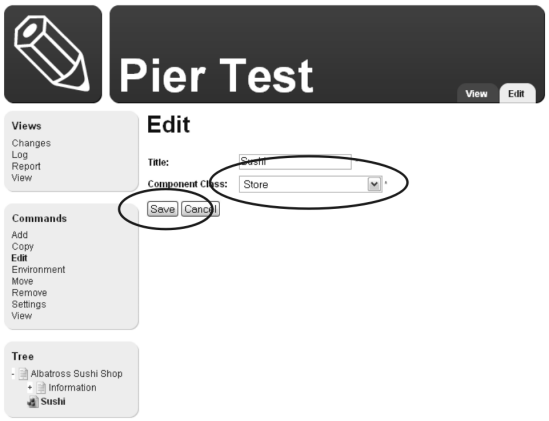
After having created the Sushi shop, we click on the name of our root page in the navigation tree to return there (Figure 2.2 bottom right). The Sushi shop is now included in this page.



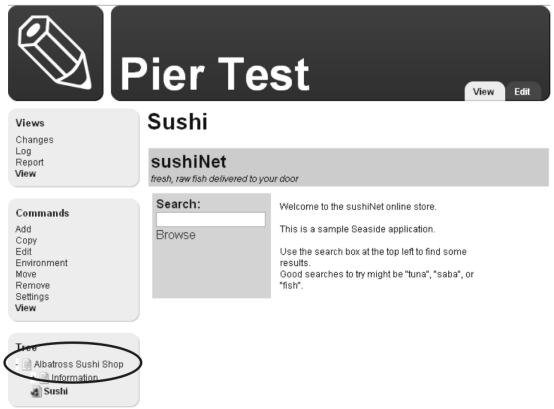

```
browser
   click: 'Sushi' in: '.contents *'.
```




```
browser
   enter: 'Component' for: 'Type';
   click: 'Save'.
```




```
browser
   enter: 'Store' for: 'Component Class';
   click: 'Save'.
```



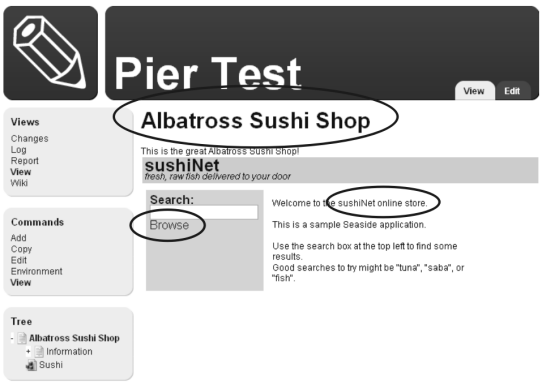

```
browser
   click: 'Albatross Sushi Shop'.
```

Figure 2.2: Pier example scenario: making a Sushi shop

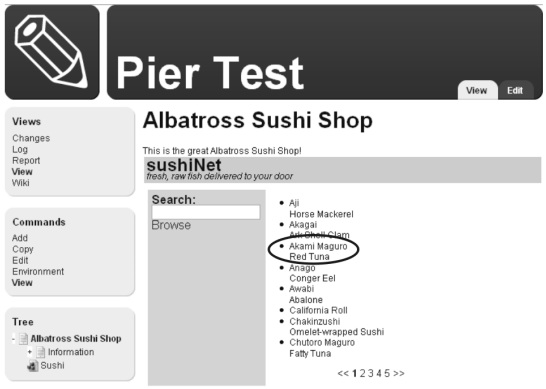
2.3 Buy Akami Maguro

To show how the direct model access works, although it is not necessary in this context, we make an assertion on the test component. We test if we are back on the 'Albatross Sushi Shop' page now. In the case of Pier, we can read the title of the current open page as shown in the first line in Figure 2.3 top left. We can check if we are on the right page by making a text assertion in the browser, too, as we see directly afterwards.

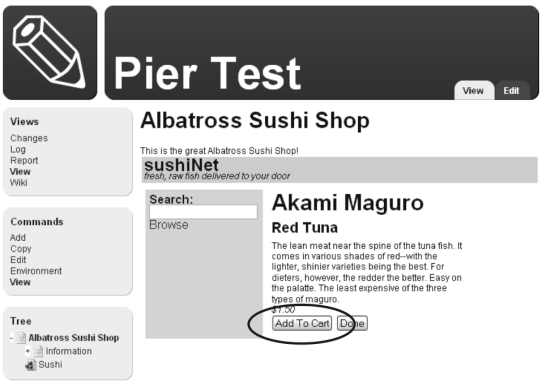
Then we want to buy an Akami Maguro. We click 'Browse' to see a list of shopping items, then select 'Akami Maguro' (Figure 2.3 top right). Then, we add it to the shopping cart. Buttons and links do not have to be differentiated to click on, so we can write `click:'Akami Maguro'` for a link and `click:'Add To Cart`' for a button. To finish, we leave the Akami Maguro page to continue shopping. We check the existence of the correct shopping cart sum that is shown in a little box (Figure 2.3).



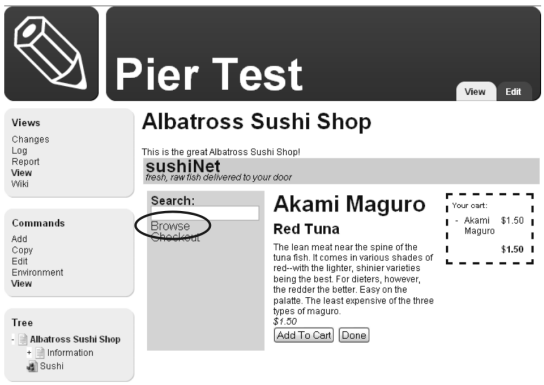

```
self assert:
   (testComponent context structure
   title = 'Albatross Sushi Shop').
self assert:
   (browser textExists: 'sushiNet online store').
browser click: 'Browse'.
```




```
browser click: 'Akami Maguro'.
```




```
browser click: 'Add To Cart';
```




```
browser click: 'Browse'.
```


```
self assert: (browser textExists: '$1.50').
```

Figure 2.3: Pier example scenario: buying an Akami Maguro

Chapter 3

Implementation

3.1 Environment

Albatross is implemented in Squeak [9]. It uses Seaside [2] and needs to run in the same Squeak image as the Seaside component under test. Albatross runs in all web browsers with JavaScript enabled.

3.2 Server - Client Communication

3.2.1 Tasks and Results

The communication between the unit test and the web browser happens by HTTP requests and responses. Each test result is attached as a field to an HTTP request. The test tasks are sent in HTTP responses in form of JavaScript commands.

Figure 3.1 shows a sequence diagram of the Albatross communication. To start the communication, the web browser is opened on the entry point `seaside/albatross`. On loading, an HTTP request is sent to the server with the field `start` set to true. An instance of the class `AModule`, which is part of the server, receives this request and adds `'start'` to the `inQueue`. The `inQueue` and the `outQueue` are shared queues between `AModule` and `ABrowser`. `ABrowser` takes `'start'` out of the `inQueue` and returns it to the TestCase to start with the test tasks. The web browser is now waiting for an HTTP response.

For each test task, `ATestCase` calls an `ABrowser` method, then `ABrowser` creates a JavaScript function to fulfill the task. Usually, it includes the command to send the result back in a new HTTP request (except when the test task triggers loading a page). `ABrowser` puts the JavaScript function into the `outQueue`, where `AModule` takes it out. It sends the JavaScript as an HTTP response to the web browser which has been waiting for this. The web browser executes the JavaScript. That can mean entering a value, following a link etc.

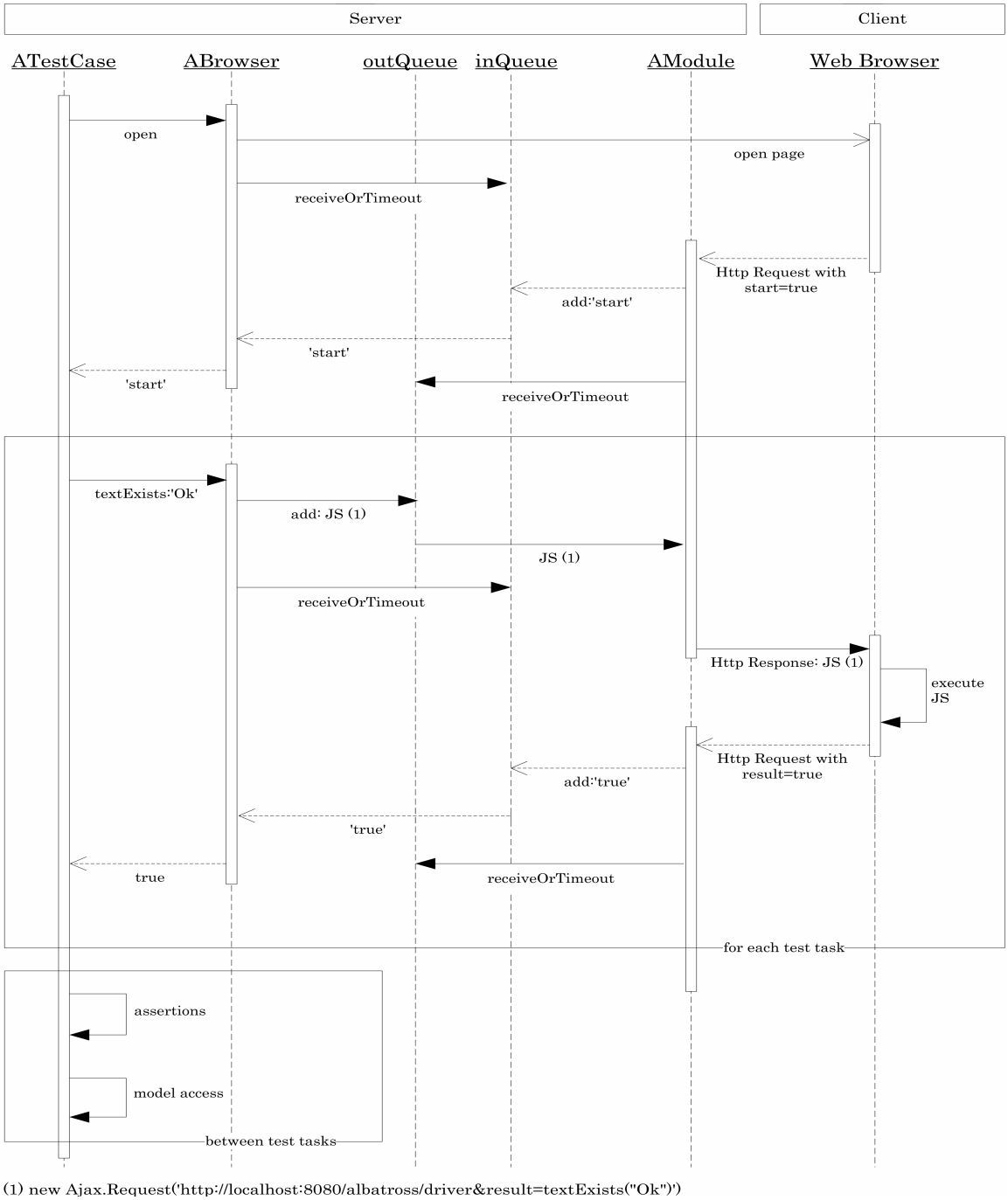


Figure 3.1: Server-Client-Communication

A new HTTP request is sent either because this was part of the JavaScript or because a new page is loaded. The HTTP request includes a field result containing a return value (or, in case of a page load, again the start field). `AModule` again receives the HTTP request and puts the result into the `inQueue`. After that, it waits for another test task in the `outQueue`. `ABrowser` takes the result out of the `inQueue` and returns it to `ATestCase`. `ATestCase` can immediately send the next test task, or it can make assertions on the result it has received or on the model of the application. `AModule` and the web browser wait for a new test task in this time.

At the end of the test, `ABrowser` puts 'end' into the `outQueue` and `AModule` sends an empty HTTP response to the web browser.

3.2.2 Test Setup

`ATestCase`, subclass of SUnitTest, creates an instance of the Seaside component under test and passes it to the AComponentWrapper. The instance creation is done in `APierTest>> createTestComponent`. `AComponentWrapper` renders the test component as its only child and appends it by the Albatross and the Prototype JavaScript. It also attaches a script that sends a first HTTP request to the server when the page is loaded. `AComponentWrapper` is registered as a Seaside application named albatross (`http://localhost:<port>/seaside/albatross`). `ATestCase` opens this application in the standard web browser of the system.

3.3 Albatross Test Messages

This section gives an overview over the API available for Albatross tests.

3.3.1 Actions

Action messages tell the browser to simulate user interactions: clicking, writing text etc.

click: clickLocal: click:in: clickLocal:in:

`click:'Edit'` clicks on the first link or button that contains 'Edit'. If there is no new page loaded as a result, we use `clickLocal:` instead. This is because the simple click: triggers no HTTP request because the page load produces the request already. If no page is loaded, the loop stops. `click:'Sushi' in:'.contents *'` specifies the elements that are searched for the wanted link or button. We use a CSS selector [11] for this, in this case for all elements that are nested into an element with the class 'contents'. Here again, `clickLocal:in:` is used for links that do not load a new page.

enter:for: enter:into:

`enter:'Component' for:'Type'` enters the value 'Component' into the form field that belongs to the element that contains 'Type'. This method is used for all form elements that have a text label near them. In the case there is no text, the element can be specified manually. To do this, methods for finding elements can be nested into `enter:into:` as explained in section 3.3.3.

back, forward

`back` and `forward` simulate the browsers back and forward buttons.

sendJS: askJS:

With these two messages, user-defined JavaScript code can be written. It is executed on the client and may return a result back to the unit test. Use sendJS: for actions that load a new page and askJS: if no new page is loaded.

3.3.2 Tests

Test messages return values that may be used for assertions.

textExists: textExists:in:

This message is sent to ask if a given string exists on the page. With `textExists: 'Hello' in: '.contents *'`, the text is searched in a given CSS selection.

title, url

`title` returns the title of the page, `url` the URL.

valueFor: valueOfId: valueOf:

`valueFor:aString` returns the value of the form field belonging to the first element that contains the given string. This is the correspondent method to `enter:for:`. In `valueOfId:` we give directly the id of the form field. This is not often used because there are better ways than ids to access elements. The value of any element can be read using `valueOf:`. In `valueOf:`, the argument is an expression built with finding-elements-functions, as explained in section 3.3.3.

idOf:

This method finds out the id of any element, if it has one.

3.3.3 Finding Elements

Sometimes we want to apply one of the above methods for an element that we want to select separately. To provide various ways of accessing elements, there are the finding-elements-messages. They do not send a complete task to the browser but only create parts that can be nested into other methods.

In many cases, there is no need for these methods. But they are needed in special cases. For example, we imagine we have three links without text, only pictures. We want to follow the second link. With some text in the link, `click:'linktext'` would be used. But now, we can for example define that we want to click exactly the second link on the page. This would result in the following code line:

```
browser click:  (browser get:2 in:  (browser css:'a')).
```

css:

`css:aCssSelector` takes a CSS selector [11] as an argument and returns an array with all matching elements. In the above example, `css:'a'` results in an array of all links of the current page.

get:in:

With `get:n in:anArray`, the n^{th} element of the array is returned. The first element has the index 1.

find: find:in: firstFind: firstFind:in:

`find:aString` searches the whole page for the given string and returns an array with elements containing that string. To search only in an array of elements, `find:aString in:anArray` is used.

`firstFind:` and `firstFind:in:` do exactly the same but return only the first element of the array, as this is very often used.

clickableIn:

`clickableIn:anArray` selects all links and buttons in the array.

id:

`id:` returns the element with the given id.

3.3.4 Model Access

Until now, we have discussed the possibilities to interact with the rendered web application in the browser. On the other side, we have the model. There is unlimited access to the model available by the test component accessor `ATestCase>>testComponent`. Depending on the test components code, we may access the model with `self testComponent model` or likewise. The model access can be useful for assertions before and after user interactions.

3.4 Albatross JavaScript Functionality

The JavaScript libraries provide the possibilities of access-by-text, intelligent handling of form values and much more. The Prototype library [10] is a JavaScript framework by Sam Stephenson. The Albatross library is part of this work.

3.4.1 Prototype

The most important method of Prototype for Albatross is explained here.

$$

`$$(css-selector)` returns all elements of the DOM tree that match the css-selector argument. Examples:

- `$$('div.content')` returns all `<div>` elements of the class content
- `$$('div.content *')` returns all elements in `<div>` elements of the class content
- `$$('input[type=text]')` returns all text input elements.

3.4.2 Albatross js

find

Albatross lets the test programmer work by what he sees: the method `find('Hello')` returns an array with all elements that display the string 'Hello'. The programmer can forget about tags, ids and names. There is one case where this method can be misunderstood: Suppose we have `<p>Hello world! Hello <b>world</b>! </p>`. Then, the command `find('world')` will return only the `<b>` element instead of the whole `<p>` element. The programmer would have to use `find('Hello')` if he wanted the `<p>`.

inputNear

This method is used to find a form field for a given label. For example, if we want to access the text input field that belongs to the text 'Name:', we call `inputNear('Name:')`. It is not trivial to find the right form field for a text label. In the simplest case, input fields stand right behind their label. There we could just take the first form element that comes after our text. But in cases as in Figure 3.2, labels stand above their input field and they are arranged in two columns. If we would simply take the first input field after the label '*Last Name', the left input field would be returned, which belongs to '*First Name'. There are even more complex cases, where form fields are on the left side of their label or above it. To provide easy access to form fields in a wide area of cases, this method makes use of some non-trivial heuristics.

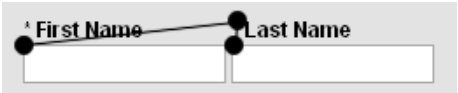


Figure 3.2: Form fields in two columns: a geometric approach is needed

First, the element containing the given label text will be searched for, then all form fields are browsed and the best matching is returned. Best matching is calculated by the following two metrics:

- Extended Euclidian distance: This calculates the distance between two elements using the Prototype `Position.cumulativeOffset(element)` which is the absolute position of an element on a web page. The space is streched in a vertical direction to prefer elements that are on the same line over elements that are below or above each other. Figure 3.3 shows how horizontal preference works. Without horizontal preference, the correct last name field is about twice as far away from the last name label as the postal code field. But with horizontal compression, their distance is about the same (See Figure 3.3 right). Together with the extended DOM sequence distance, the correct field is chosen.

Figure 3.3: Horizontal compression to give importance to elements of the same line

- Extended DOM sequence distance: This calculates the distance of two elements in the linearised DOM tree. That means it uses the elements sequence in the HTML

document: in Figure 3.3, the 'Last Name' field has distance 1 to the 'Last Name' label, the 'Postal Code' field is at least 3 elements away. Additionally, the distance is stretched when a is after b and made smaller when a is before b. By doing this, we can achieve that a form field that is on the right side or below the label is preferred to a field on the left or on the top of the label. This is important in cases as seen in Figure 3.4: here, the 'Last Name' input field is geometrically nearer to the 'Enter some little text here' label than the field at the bottom. And they are both one DOM element away. With the stretching of the DOM sequence distance, the correct field is chosen.

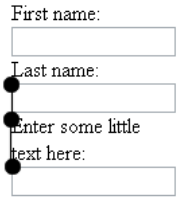


Figure 3.4: Preference of form fields after the label over those before

In our experience, in most cases, the proper form field has been found. If we want to specify the label element by hand, we can use `inputFor(element)` directly. We need to do this if there is no label text or if for any other reason, the input field is not found correctly.

click

Clicks the first link or button that contains the given text. Or if we use an element as argument, this element is clicked.

value

Returns the displayed value of any kind of elements, expecially of all form elements. For select boxes, it returns directly the selected text instead of an id.

enterValue

Enters values in any kind of form elements. Again, we never use ids but meaningful content.

Chapter 4

Conclusion

Albatross wants to encourage Seaside programmers to do test driven programming. It is a little package to download into a Squeak image and provides a straightforward and fast way to write and run test scenarios.

The Albatross interface provides good support for the often used actions like following links, clicking buttons and inserting values into form fields. They all require only some corresponding text to identify the programmers wish; ids, names and HTML tags do not have to be specified. For more complex situations, there are a lot of ways to still express a desired task easily by simply nesting some methods. Even writing own JavaScript code is supported.

Writing the Pier example test was easy. Most of the required actions could be typed directly in the test. The only problem was to click the link 'Sushi', because there was a link 'Albatross Sushi Shop' that was previous to it on the same page (See section 2.2). There, a solution had to be found to select the correct link, which resulted in specifying a CSS selector to say that we wanted the link in the 'contents' area. This idea does not fully support the idea of forgetting about ids and names, but it may be convenient to the programmer. It makes use of the few important ids the programmer set himself purposely.

Further development could introduce a mechanism to automatically record a scenario in the browser. Another approach could be to coordinate with the Seaside Testing Framework [7] to create a full Seaside testing suite. With Albatross, all JavaScript enabled web browsers can be used to directly test the application, including Ajax and JavaScript behaviour.

Bibliography

[1] Ducasse, Stéphane. *SUnit Explained*. http://www.iam.unibe.ch/~ducasse/Programmez/OnTheWeb/SUnitEnglish2.pdf

[2] Ducasse, Stéphane ; Lienhard, Adrian ; Renggli, Lukas: Seaside — a Multiple Control Flow Web Application Framework. In: *Proceedings of ESUG International Smalltalk Conference 2004*, 2004, S. 231–257

[3] Garrett, Jesse J. *Ajax: A New Approach to Web Applications*. http://www.adaptivepath.com/publications/essays/archives/000385.php

[4] Goldberg, Adele ; Robson, David: *Smalltalk 80: the Language and its Implementation*. Addison Wesley, May 1983

[5] OpenQA. *Selenium Remote Control*. http://www.openqa.org/selenium-rc/

[6] Renggli, Lukas: *Magritte – Meta-Described Web Application Development*, University of Bern, Diplomarbeit, Juni 2006

[7] Shaffer, C. D. *Seaside Testing Framework*. http://www.shaffer-consulting.com/david/Seaside/SeasideTesting

[8] Shaffer, C. D.: *Squelenium*. http://lists.squeakfoundation.org/pipermail/seaside/2005-July/005473.html. – Selenium driver for squeak

[9] *Squeak*. http://www.squeak.org

[10] Stephenson, Sam. *Prototype*. http://prototype.conio.net

[11] W3C. *Cascading Style Sheets*. http://www.w3.org/Style/CSS/