WebAppQA

A Language for Testing Web Applications

Adrian Frei (af2364)
Ankit Malhotra (am2994)
Peter Lu (yl2505)
Roy H. Han (rhh2109)

Submitted as a final project proposal for
COMS W4115: Programming Languages and Translators

Taught by
Professor Stephen A. Edwards

Introduction

WebAppQA is a language that simplifies the task of writing unit tests for a web application. It is a mixture between a regular scripting language like Perl or Python, unit-testing frameworks like jUnit and tools like BadBoy. With WebAppQA, users write tests to check page format, functionality and security. The interpreter runs each test and produces a report.

- Existence – Do all the links on a page work?
- Content – Does a page match a regular expression?
- Mechanics – Are links and cookies well formed?
- Security – Is the page safe against cross-site scripting attacks?

Usage Scenarios

1. A web shop wants to automatically check a number of its product pages. Several salespeople constantly modify these pages and sometimes links are broken and images and offers are missing.
2. A company develops software in small iterations in an agile manner. This means that the production system is updated frequently several times a day. Unfortunately this also means that the chance of errors on some pages is quite high.
3. Wikis and other user-defined content management systems are volatile and prone to error. Tests written with WebAppQA can provide immediate feedback to the contributor before the page is released to the public.

Language Details

The language is dynamically typed and runs in an interpreter. It supports loops, conditions, variables, methods with a syntax similar to Ruby. However there are no user-defined classes, iterators or blocks.

Data Types

All data types are objects and WebAppQA supports the following basic types: integers, strings, regular arrays and hash tables.

More data types can be added as classes as part of libraries. At this time there is one such class: the Site class. Objects of this type support a number of methods to visit, navigate and test web sites.

The language is dynamically typed without need for declarations. It is possible to have global variables (@ prefix) and local variables (no prefix).

Operators and Methods

Integer

- + - * / % : Addition, Subtraction, Multiplication, Division, Modulo

String

- + : Concatenation with other strings and integers
- \\ : Regular expressions

Array

- array[i] : Array lookup

Hash Table

- hash['key'] : Hash table lookup

Site Class

Constructor:

- Site.new(url, arg, method): Creates a new object and visits the site

Methods:

- visit(url, arg, method): Visits a page

Attributes:

- status: The status of the page (e.g. 404 or 200)
- links: All links of a page
- images: All images of a page
- contentType: The content type (e.g. text/html)
- content: Content of the page as a string
- cookies: The cookies set by the page

Control Flow and Other Keywords

The language supports for-loops and while-loops, if and else conditions. Users defined methods using the keyword def.

The *test* keyword defines a test in a manner similar to the def construct.

The *setup* keyword defines a test that is executed before all other tests.

The *assert* keyword defines an assertion that needs to be true for the test to succeed. If an assertion evaluates to false, the execution is either interrupted or not, depending on the keyword following it: fail or warn respectively. In either case, the error is added to the test report. The assert construct is similar to the unless construct in other languages.

Following *fail* or *warn* it is possible to specify an expression that will appear in the report. Otherwise, in the absence of any such statement, the default behavior is to print the stack trace of the failed assertion (see Report Example).

```
# Fail with assertion evaluates to false
assert site.status < 400 fail
# Warn the user if assertion is false but continue
assert site.status < 300
warn "Site status must be smaller 300, got" + site.status
```

Another construct is the *with* keyword. It is syntactic sugar that allows a developer to type less when accessing methods or attributes of objects. The with keyword shadows local variables and functions with the attributes and methods of an object.

```
object.method1()
object.attribute1 = 1
object.method2()
```

can be written as

```
with object
   method1()
   attribute1 = 1
   method2()
end
```

Code Example

The example below tests a small web shop.

- Test that the home page exists
- Test the login and make sure cookies are set and the user is forwarded to the welcome page
- Test that the website verifies cookies and allows access to logged in users.
- Test a number of offer pages and makes sure that the pages display the merchant's logo and that all links on the page are working

```
# Define a setup function (called first)
setup initialize()
    # Global string
    @startURL = "www.myshop.ch"
    # Global hash table containing string
    @loginParam = {"username" => "hans", "password" => "p8ssw0rd"}
end

# Define a test function
test home()
    # Create a new site object and visit the site
    site = Site.new(@startURL)
    # Make sure the page is there
    assert site.status == 200 fail
end
```

```
test login()
    # Visit a site with parameters with a POST request
    site = Site.new(@startURL+"/login", @loginParam, "POST")

    with site
        assert status == 200 fail
        # Only accept html content
        assert contentType == "text/html" fail
        # Page must contain Welcome title
        assert content ~= "/<title>Welcome</title>/" warn

        # Make sure the fields visits and id are set
        assert cookie["visits"] > 0 warn
        assert cookie["id"] != nil warn
    end
end

test modifyCookie()
    # Log into the site
    site = Site.new(@startURL+"/login", @loginParam, "POST")
    # Make sure the id is set
    assert site.cookie["id"] != nil fail
    # Modify id
    site.cookie["id"] = "deadbeefcafe"
    # Browse to user setting
    site.visit(@startURL+"/setting")
    # Must be redirected to home page
    assert site.status = 302 fail
end

test odp()
    # Define array with page numbers to visit
    pagesNr = [1,4,6,12,15,102]
    # Loop through all pages
    for i in 0..pagesNr.length
        # Construct a url with the page number
        url = @startURL+"/odp/"+pagesNr[i]
        site = Site.new(url)
        assert site.status == 200 fail
        # Make sure the image is there
        # Page number > 100 have a different image
        if i < 100
           assert Site.new(url+"/logo.jpg").status == 200 warn "Failed at " + i
        else
           assert Site.new(url+"/logo_big.jpg").status == 200 warn "Failed at " + i
        end
        # Call function to make sure all links are alright
        linkCheck(site)
    end
end
```

```
def linkCheck(site)
    links = site.links
    for i in 0..links.length
        assert Site.new(links[i]).status == 200 warn
    end
end
```

Report Example

```
=== Report ===

== File ==
  example.qa

== Summary ==

Test(s) failed
  * login
  * odp

== Tests ==

= initialize =
  Success

= home =
  Success

= login =
  Failed
    assert content ~= "/<title>Welcome</title>/" warn
      "<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 St..."
    assert cookie["id"] != nil warn
      {visits => 12}
      nil

= modifyCookie =
  Success

= odp =
  Failed
  assert Site.new(url+"/logo.jpg").status == 200 warn
    "Failed at 12"
  assert Site.new(url+"/logo_big.jpg").status == 200 warn
    "Failed at 101"

  - linkCheck -
    assert Site.new(links[i]).status == 200 warn
      "www.myshop.ch/odp/123123/logo.jpg"
      Site.obj#1243141
      300
```