

Software Testing Framework

Shumoku 種目

Hitarth Patel, Richard Lee, Kristen Kho

May 29, 2008
CSE 210

Introduction/Overview

Based on the Model-view-controller (MVC) pattern described in our Software Design Spec, we divided up our testing into the same three layers. Our three layer testing will allow each component to be unit tested separately. This also helps narrow down potential bugs can have occurs at any layer. We also analyze the various risks to our project currently and how we have used our current testing framework to minimize these risks.

UI Interface (View)

Our UI Interface is tested using Watir -- a simple open-source library for automating web browsers. Watir stands for "Web Application Testing in Ruby". Watir is a Ruby library that works with Internet Explorer on Windows. It allows you to write tests that are easy to read and easy to maintain. It is optimized for simplicity and flexibility. Watir drives browsers the same way people do. It clicks links, fills in forms, presses buttons. Watir also checks results, such as whether expected text appears on the page.

We are in the process of writing a library of Watir scripts that test the functionality on the Shumoku website. They are designed to test form elements and form submission and verify that the server returns the expected results. Figure 1 is an example of a script written using Ruby/Watir. It simply clicks on the "Find Events" button and checks that the page returned contains the string "Your search returned".

```
#-----#
# shumokuSearch001.rb
# Simple test for Find Events button.
#
# author: Kristen Kho
# date: 05.29.08
#-----#

require 'watir' # the watir controller

testName = "Shumoku Search 001"
expectedResult = "Your search returned"

# set a variable
test_site = 'http://www.kirinroad.com/shumoku/main.php'

# open the IE browser
ie = Watir::IE.new

# print some comments
puts "## Beginning of test: " + testName
puts " "
```

```

puts "Step 1: go to the test site: " + test_site
ie.goto(test_site)
puts "  Action: entered " + test_site + " in the address bar."

puts "Step 2: click the 'Find Events' button"
ie.button(:name, "search_button").click # the name of the Search button
puts "  Action: clicked the Find Events button."

puts "Expected Result: '"+ expectedResult +"'"

if ie.text.include?(expectedResult)
puts "Test Passed. Found the test string: '"+ expectedResult +"'. Actual
Results match Expected Results."
else
puts "Test Failed! Could not find: '"+ expectedResult +"'"
end

puts "  "
puts "## End of test: " + testName

```

Figure 1. A UI test script written in Ruby/Watir.

Server Side Layer (Controller)

Server Side Layer will be tested using the SimpleTest PHP Framework (a framework similar to JUnit, but written for PHP). Test cases are written in a similar convention as JUnit test cases with output in either HTML or standard output. Additional scripts will be added as the development comes along. As part of our process, the scripts will be run before and after integration to help stabilize our code on the server-side layer. Future extension of this framework can include automated testing during checkins to source control and nightly builds.

Database Layer (Model)

Database tables will be tested using MySQL and PHP Framework(similar to Server Side layer). Testing will include connecting to a database and storing and retrieving events and event log information. In the 1st release of Shumoku, we are not including user accounts.

Risks

1. Use case scenarios presented during the demo may blow up.
2. Integration may cause functionality to break.
3. New components may be added.
4. Unintended actions by user not handled gracefully.
5. Miscommunication between layers (passing invalid parameters, returning wrong results).
6. Database connection may fail during the demo.
7. Database script to remove old events may fail (This will corrupt the results of search queries with invalid event information).

8. Performance issues (site/network traffic).

Risk Resolution

1. UI layer testing will test all demo use cases. We will run them before our demonstration, so chance of a complete blow up is minimal. In a
2. Since our testing is layered, we are able to do unit testing before and after integration. We could have just tested the UI layer, but potential bugs could be hidden and debugging would be difficult. (On a side note, a company that one of us worked for did their unit testing this way).
3. As we add more components (or extend existing components), we have a larger opportunity for our code to break. This is a larger risk when doing integration on features that cross two or more layers. The test suite on each layer allows us to test new code immediately to minimize this risk.
4. We will write as many test cases as possible to attempt to capture all possible scenarios. Getting feedback from actual users will help to discover what these scenarios are.
5. Each layer's documentation will be kept updated with the latest changes and team members will be informed of such changes to be able to update the calling interface.
6. Database testing will be done at each level of integration (Database drivers, Server side, and UI).
7. The database script will be run manually to verify that it removes past events, and it will be automated in a future release.
8. As Shumoku gains a larger audience, performance may become a large issue. In the future, performance and load testing with tools like JMeter will be considered. They will help us decide whether we need to upgrade our hosting plan or purchase a dedicated server. However, since our product is still in a state of infancy, we may defer these decisions until later.

Risk Resolution Results

1. More extensive UI testing on common scenarios, especially those that will appear in the demo.
2. Addition of unit testing before and after check-ins (similar to Microsoft's approach) into our software development process. This will stabilize our code.
3. We have taken the test-first approach (Agile and XP) in developing our code. While implementing features, we are also adding test cases to test the new features (and testing whether we broke old test cases).
4. Prioritization of tests for important user scenarios.
5. Addition of documentation and commenting standards. Fortunately, Eclipse automates a comment template for each function, so the developer only needs to fill in the blanks.
6. If remote database connection fails, we will demo the product using a local database instance and an Apache server on a local machine.
7. From UI, events will be searched using past dates to verify that past events are removed from database.
8. Performance and load testing suite development are deferred for future releases. We will consider with our current hosting plan and server.

Commitment

- Demonstration to Small World Venture Capital on June 9th.
- Maintain current testing framework.