An Object-Oriented Architecture Supporting Web Application Testing

Ji-Tzay Yang, Jiun-Long Huang, Feng-Jian Wang
{jjyang,jlhuang,fjwang}@csie.nctu.edu.tw

Computer Science and Information Engineering
National Chiaotung University
Hsinchu City , Taiwan 300

&

William. C. Chu
chu@cis.thu.edu.tw

Computer Information Science
Tunghai University
Taichung City, Taiwan 400

Abstract

The flexibility and rich application frameworks of Web model make Web applications more prevalent in both Internet and Intranet environments. Programmers enjoy various of Web application frameworks whose support ranging from simple user interactions based on plain client-server model, to complicated distributed-object computations based on CORBA. The variety gives user the flexibility to decide a proper framework, and leads to the demands of new supporting tools and testing framework to test and maintain Web applications. This paper presents an architecture containing several supporting tools which enhance traditional software testing architecture to fit common Web application frameworks. The architecture suits current Web models and reuses several software patterns and architectures from traditional testing environments. In addition, a prototype Web application testing environment is constructed for demonstration.

1. Introduction

Web model and its related improvement give Web application designers flexibility at choosing proper development products for their implementation. Current developers of large Web application do not have sufficient and powerful tools to debug or test their Web applications. Existing Web testing tools on Internet are usually made for verifying the syntax in HTML documents, checking the hyperlink integrity in a set of HTML documents, testing GUI components embedded in browsers, and measuring the performance of the Web application. Few products support overall Web applications testing. [10] and [11] test the software components such as Java Applet and ActiveX objects which are embedded in the Web pages. [6] helps justify the result shown on the Web browser's window by matching text patterns or pixel-level comparison. [8] checks the documents for syntax and compatibility to popular Web browsers.

The paper presents a software architecture integrates several conventional testing tools. The architecture extends these traditional software testing architectures and software patterns [5] [9] to ease the description and design of Web-application testing tasks. The integration of Web-testing components can reduce the insufficiency of independent Web testing tools mentioned above for complicated Web application testing. Constructing a testing environment for Web applications to demonstrates the reuse of the software architecture. With object-oriented technique, the architecture itself provides a clear picture of software components for reuse including tool reuse, architecture reuse, etc.

This rest of the paper is organized as follows. Section 2 discusses the issues of Web application testing based on traditional testing techniques and popular Web application development models. Section 3 presents the software architecture which accommodates the tools to Web software testing environment. Section 4 demonstrates a prototyped testing environment constructed under the architecture. Finally, section 5 gives a concluding remark.

2. Specializing Application Architectures to Test Web Application

2.1 Software Architectures for Software Testing

The architectures for software testing environment have been studies for years. Traditional software testing environments are evaluated and divided into the following five subsystems [2] [7]: (1) test development, (2) test measurement, (3) test execution, (4) test failure analysis, and (5) test management. [1] summarizes the software architecture reuse approaches in several aspects such as *abstraction* and *integration*. *Abstraction* comes from application domain and is realized by defining objects and their operations in domain language while *integration* is to integrate different domain components into a single application.

A testing environment for Web applications can be viewed as a specialization of traditional software testing environment. The architecture which supports testing of Web applications can be obtained from existing architecture of software testing environment, by specializing the architecture and introducing some new domain components which support the testing tasks for Web applications. Traditional software testing architecture has done well in dividing a testing system into subsystems recursively till domain primitives. When extending the Web testing architecture from traditional one, Both the *abstraction* and *integration* aspects can be considered. In the aspect of *abstraction*, the effort might be devoted to construct Web-related domain components (primitives) which help to describe Web-related testing operations. In the *integration* aspect, an object-oriented approach for the architecture is suitable for integrating domain components to perform specific tasks in the testing environment.

2.2 Constituents of Web Applications

Before discussing how to test Web applications, Web model

0-7695-0368-3/99 $10.00 © 1999 IEEE

provides application platforms or application designers with several locations to place code for Web computation and alternative mechanisms to solve particular missions. Figure 1 depicts the typical constituents of the Web application.

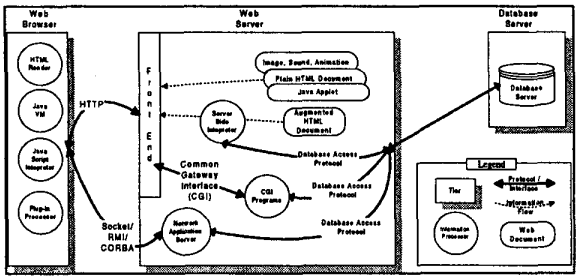

Figure 1. Constituents of typical Web applications

The *Web browser* is capable of retrieving hyper-text documents, as requested by the application users, from the Web server via HTTP protocol. It renders the hyper-text document in HTML (Hyper-Text Markup Language) format on the screen. Contemporary Web browsers also embed Java virtual machine and Java Script interpreter to execute the Java Applets or Java Scripts specified in the documents. Additional information processor such as Netscape Communicator's *plug-in* modules and Microsoft Explorer's ActiveX objects, which are browser-loadable software modules, can extend browser's functionality.

HTTP daemon is placed at the Web server to accept the HTTP requests from the browsers. According to Web server's configuration, it may forward the request to (1) document retriever for serving stored HTML documents, Java Applets, or multimedia files, or (2) to other information processor on the Web server, such as CGI programs for dynamically generated HTML documents and contents. Web servers are sometimes equipped with information processor, e.g. Apache Module, or Active Server Page Engine to perform the computation defined in augmented HTML documents before sending them to browsers.

Protocols convey command, document or executable between *information processor*. *HTTP* is used for communication between Web browsers and Web servers. The *CGI* is a standard for external gateway programs to interface with information servers such as HTTP servers. It forwards both input data and output HTML document for Web browser and CGI programs running on Web server. *De facto* database access interface such as *ODBC* or *JDBC* connect Web servers and database servers. Communication between roles of Web application components may also flow through *plain TCP sockets, Java RMI, or CORBA*. These emerging protocols are more suitable for developing distributed Web application in object-oriented technologies.

The placement of constituents in Web model can divide Web application constituents into three major tiers: Web browser tier, Web server tier, and database server tier. The information process in the application is passed through each tier. The user interaction is performed at the Web-browser tier. The program logic computation is performed at the Web server tier. The database operation is done at the database-server tier. Hence, the Web application model is also known as a *three-tier* application architecture. When the database server tier is omitted, it is known as a *two-tier* application model.

2.3 Domain Components for Web Application Testing

Several approaches have been selectively used by Web application developers to construct Web applications according to user interaction and program logic. Followings are typical and combinable scenarios in Web application construction:

(1) Applications which consist of augmented HTML documents: Augmented HTML documents are processed at server-side by processor ranging from macro processor (e.g. Apache's server-side include) to embedded script (e.g. Microsoft Active Server Page, Server-side JavaScript).
(2) Applications that contain scripts running at client-side: Scripts written in JavaScript or VB Script are executed by the browser to perform user interaction and data validation.
(3) Applications which are originally developed in traditional languages such as C++ or Perl, and interact with Web client through CGI: Legacy applications equip themselves with Web features by adding modules to receive requests and reply results through CGI (or its derivative – ISAPI and NSAPI).
(4) Applications that apply HTTP-cookie to implement session-semantic in Web environments: cookies are sent back and forth between Web server and browser to track session status and variables.
(5) Applications that connect to database servers: Applications on Web servers mainly use SQL statements and ODBC to communicate with database servers.

To perform Web application testing with respect to above application scenarios, domain components might be included to testing environment as the primitives, to help describe testing tasks. The domain components (tools) suggested by authors are listed in terms of subsystems of the Web application testing architecture and shown in section 3.

3. Architecture of Web Testing Environment

[2] proposed an architecture for traditional software testing environments, and it is well evaluated in [7]. We extend this architecture for testing of Web applications testing as described in this section.

3.1 The Architecture

According to the architecture in [2], a software testing environment consists of five subsystems. With the growth of Web application techniques, more and more Web programming styles (e.g. ASP, JavaScript) have been proposed. These programming styles introduce several new techniques which were not used in conventional software. For example, one document may contain several code fragments written in different programming languages, and these fragments may be interpreted in different tiers such as browser, server, database, ...etc. Therefore, one Web application should be analyzed at browser, server, even database tire, and the corresponding analysis services for different programming languages are also needed. We add a new subsystem named Source Document Analysis into above architecture to handle the testing problems introduced by these new programming styles. Figure 2 shows an overview of this architecture, where solid lines indicate data flow.

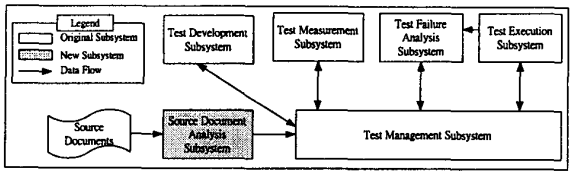

Figure 2. Architecture of Web Application Testing Environment

3.2 Source Document Analysis Subsystem, SDAS

Different programming approaches applied for Web application developments have different characteristics. For example, server side programming is focused on database accesses and able to generate documents to Web browser according to the result of database queries. Client side programming is focused on GUI representation and manipulation in Web browser. A programming approach may need a distinct programming language, which is associated with a set of tools, such as *Server-Side Script Interpreter*, *Client-Side Script Interpreter*, *HTML Analyzer*, and so forth.

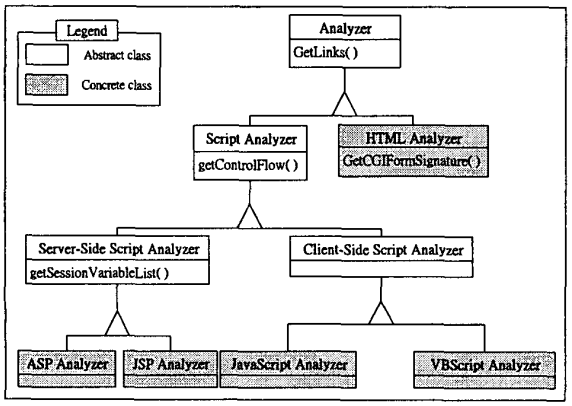

Figure 3. The Class Diagram of Source Document Analysis Subsystem

Figure 3 is the class diagram of these tools. All tools designed to analyze the source documents and extract some information (e.g. hyperlink) are called *Analyzer*. There are two categories of *Analyzers* for Internet software developed, *HTML Analyzer* and *Script Analyzer*. *HTML Analyzer* processes HTML fragments in source documents to extract information such as CGI From. *Script Analyzer* is used to process the embedded script fragments and extract information such as control flow of these script fragments. *Script Analyzer* is divided into two categories, *Client-Side Script Analyzer* and *Server-Side Script Analyzer*, according to the location which the script fragments are interpreted. *Client-Side Script Analyzer* handles the script fragments which are interpreted in Web browser, and *Server-Side Script Analyzer* handles the script fragments which interpreted in Web server.

1. SDAS extracts information such as control flow from source documents, and sends them to TMS. Control flow is useful in software testing. [4] and [3] proposed methods to construct control flow of Web applications based on the hyperlink relations between source documents. A source document may contain HTML, server-side script, and client-side script at the same time. Analyzers are designed to extract hyperlinks in three parts respectively.

3.3 Test Management Subsystem, TMS

Testing (and validation) deals with many artifacts which may be created during earlier development phase(s) or even validation phase [2]. Compared with traditional software, Web applications involve additional roles for Web such as Web server and browser, and additional control mechanisms such as cookie and session. Therefore, testing on Web applications is different (and might be more complicated than) on traditional software, and the test artifact management (e.g. manipulation of test cases) is more important. TMS works as the warehouse of other subsystem to provide testing artifact management. It contains *Application Information Repository* and *Test Suite/Case Repository*, where each repository has its own manager to handle repository manipulation.

A testing model [3] is used to describe some behavior of Web applications and the corresponding information such as control flow and data flow is stored in *Application Information Repository*. *Test Suite/Case Repository* stores test suites (and cases) which include test data, execution path, execution result, test report, and so forth. TMS provides a set of repository access interfaces and separates the interfaces from their implementations. With these access interfaces, other subsystems can create, manipulate, delete and query Repositories without concerning their implementations. In TMS, Manager pattern [9] is applied to achieve this goal.

Figure 4 is the architecture of TMS based on Manager pattern. To manipulate test suites or cases, the client subsystem sends request messages to *Test Suite/Case Manager*, which then loads or creates the *Test Case* objects correspondingly. After manipulation, client subsystem can use the Test Case object directly. Since the client subsystem is not related to the implementation of *Test Suite/Case Repository*, the former does not need modification when testers change the implementation of the latter. The implementation of *Application Information Manager* is similar to that of *Test Suite/Case Manager*.

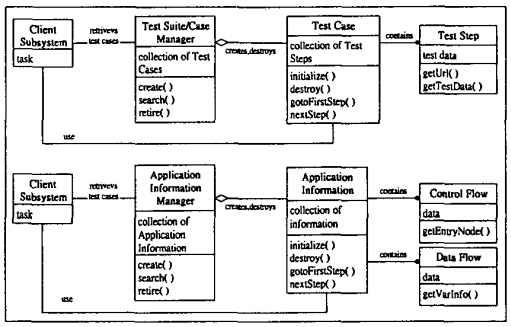

Figure 4. Test Management Subsystem

3.4 Test Development Subsystem, TDS

TDS allows testers to manipulate test suites and cases by generating creates test cases automatically, or constructing them manually based on application information generated from SDAS. After receiving the user's instructions, TDS will send these requests to TMS to execute these requests.

TDS contains five tools, *Test Case Generator* (TCG), *Test Suite/Case Maintenance Tool* (TSCMT), *Test Case Recorder* (TCR), *Test Case Composer* (TCC), and *Test Case Viewer* (TCV). For a Web application, TCG generates test cases automatically under a testing criterion such as all-statements, all-branches coverage, ...etc. *Test Data Definition Grammar* is a context-free grammar, which is designed for testers to describe the specification test data. Test data for these test cases are generated based on specification. Figure 5 shows an example fragment of *Test Data Definition Grammar*. All non-terminal symbols are starting with $. In this example, the test data of `$StudentID` can be generated as `u8617535`, `U8217045`, ...etc.

```
$StudentID→$IDPrefix+$Num+$Num+$Num+$Nu
    m+$Num+$Num+$Num
$IDPrefix→u|U
$Num→0|1|2|3|4|5|6|7|8|9
```

Figure 5. An Example fragment of Test Data Definition Language

There are at least two problems in automatic test data generation. The first problem is that the generated test cases may not be practical. Figure 6 is an example test model. Based on all-statement coverage criterion, three test cases in Figure 7 may be generated. In this example, the test data in `N1` may cause `N4` not to contain a hyperlink to `N5`, and the test case 3 is impractical. The second problem is that not all significant scenarios are covered by generated test cases. In this example, an execution path of `N1→N4→N3→N5` may be significant scenario that is used in real case, but it is not covered by the test cases generated.

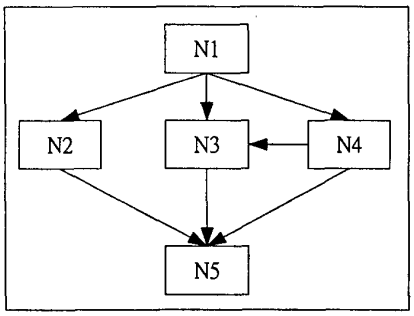

Figure 6. An Example Test Model

Test Cases No.	Execution Path
1	N1→N2→N5
2	N1→N3→N5
3	N1→N4→N5

Figure 7. Test Cases Generated from Test Model in Figure 6

TSCMT provides functions to help tester for manipulating test suites/cases to solve the first problem introduced by TCG. These functions include test suite/case editing, deleting, reviewing and so on. After test cases are generated, testers can review these test cases with TCV, a graphic tool to view the test cases, and make modification or even deletion for these test cases with TSCMT.

TCR creates test cases semi-automatically to help solve the second problem discussed above. A desired test case, which is not generated by TCG, can be created with TCR. On the other hand, tester can use Web browser to execute a Web application, and TCR can record the execution scenario and translate it to test case. There are at least two subclasses of TCR: *Data Input Sequence Recorder* (DISR) and *GUI Event Recorder* (GUIER). Figure 8 shows a class diagram of *Test Case Recorder*. DISR records the execution sequences and the data entered in CGI form by testers, and constructs test cases. DISR contains two modules, *HTTP Bridge* and *HTTP Analyzer*. *HTTP Bridge* captures the HTTP communication between Web server and browser. The captured communication is analyzed by *HTTP Analyzer* to construct test cases. The *HTTP Header Analyzer* analyzes the header of HTTP communication to extract the value of CGI form inputs. The *HTML Analyzer* analyzes the HTML part to extract the information about CGI form (e.g. the name of CGI form input). Then, *HTTP Header Analyzer* and *HTML Analyzer* send the result of analysis to *Test Case Constructor* to construct test cases.

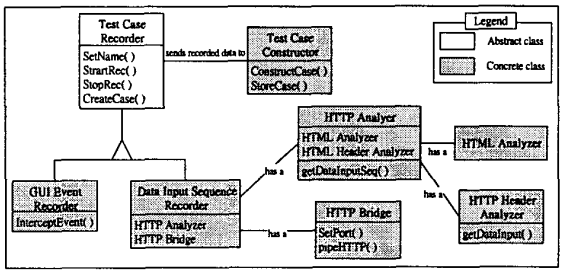

Figure 8. The Class Diagram of Test Case Recorder

[12] addressed that more applications have been built with complex graphical user interfaces (GUI), and the testing for GUI-based system is getting more important to validate the behavior of GUI. Web application is treated as a GUI-based system when the behavior of GUI is concerned only. GUIER records the user-caused GUI events about Web application (mouse movement, button clicking...) by intercepting the events between application and operating system to construct test cases. It is suitable to test ActiveX controls and Java Applets

TCC is an editing tool that provides test case modification capability, and can be used to refine existing test cases and create test cases manually. It is another solution for the second problem introduced by TCC. If testers are familiar with Web application and testing environment, they can develop test cases manually by writing test script using TCC.

3.5 Test Execution Subsystem, TCE

Test Case Executor executes test cases automatically or manually by following the parameters specified by the tester and retrieving the appropriate information from Test Suite/Case Repository. Each execution is verified by *Test Oracle* or the tester to determine whether the execution matches the specification of Web application. In conventional software testing environments, test data are filled into tested software via standard input (stdin). Web application has two kinds of test data, the user-input data and the user-caused GUI events. *Data Filler* and *GUI Event Generator* are designed to fill the test data to the Web application automatically, respectively.

```
set $URLBase http://dsslnt/webapp/
set $URL1  "login.html"
```

```
#set variables URL1, URL2
set $URL2  "checkLogin.asp"
#------- HTTP requests begin ------
HTTPGet $URL1
#login.html contains a form with two
#fields user and pass
set $form1.user  "user001"
set $form1.pass  "wrongPass"
#user001 login with wrongPass
HTTPPost $URL2, $form1
#expect an HTTP-redirect command,
#which redirect the browser
#to errorMessage.html
expect URL "errorMessage.html"
.....
press button "Reload"
press link "Registration"
move mouse 400 400
select list "File" "Open"
.....
expect FORM
Text "User Name" "Sharon"
Text "Age" "23"
.....
expect GRAPHIC "result1.bmp"
```

Figure 9. A Test Script Sample

Figure 9 shows a test script sample and the key word is represented by **bold** font. The key word `set` is used to indicate the user-input data. For example, `set $form1.user "user001"` represents that the user has entered `user001` in text field `user` with form `form1`, and the *Data Filler* will fill this text field while executing this line. Some key words such as `press` and `move` represent the user-caused events. For example, when the user press the button `Reload`, and the *GUI Event Generator* will generate a button clicked event to this button while executing this line.

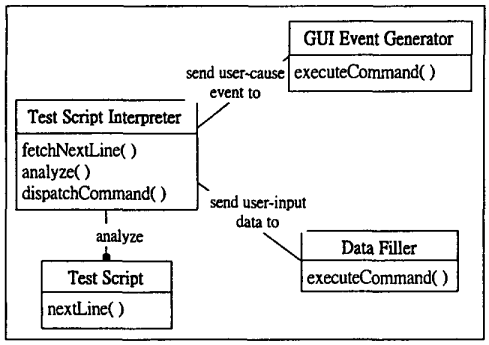

Figure 10. The Class Diagram of Test Execution Subsystem

Figure 10 shows the class diagram of TES. The *Test Script Interpreter* analyzes the *Test Script* and dispatches the commands to *Data Filler* or *GUI Event Generator* for execution. These targets are command-oriented, i.e. different commands (types) are sent to different target. For example, `press` and `move` are commands for user-caused events, and they are sent to *GUI Event Generator*.

3.6 Test Failure Analysis Subsystem, TFAS

Test Failure Analysis includes behavior validation and the analysis of test execution pass/failure statistics [7]. The *Test Oracle* analyzes the execution results of test cases, determines which of them are correct, and generates the *Test Failure Report* of these test cases. The *Test Suite Summary Generator* reads all test cases and test results belonging to one test suite, and generates *Test Suite Summary* indicates what percentage of test cases passing the test.

Expectation part in test script is used to specify the expected results of test cases, and the content in expectation part is used by *Test Oracle* to determine the correctness of results. Based on the correctness definition of test results, we summarize that there are three kinds of *Test Oracles*: *URL Oracle*, *Form Field Oracle*, and *Graphic Oracle*. *URL Oracle* assumes that one test result is correct if the URL of the returned Web page matches the expected URL of the test case. It is suitable for the static Web pages whose representation of this page will not change. *Form Field Oracle* assumes that one test result is correct if the value of each form field or table cell matches the expected value. It is suitable for the dynamic Web pages whose representation depends on the result of database queries. *Graphic Oracle* assumes that one test result is correct if the graphical representation matches the expected one. It is the strictest definition and generally is done by pixel comparison, and is suitable for ActiveX and Java Applet testing.

In Figure 9, there are three distinct expectation parts starting with key word `expect`. The first expectation *URL Oracle* part indicates that after executing `HTTPPost` command, the URL of returned Web page is `errorMessage.html`. The second expectation *Form Field Oracle* part indicates that the returned page contains two CGI form inputs with `Text` type. Besides, their names are `User Name` and `Age`, and their values are `Sharon` and `23`, respectively. The third expectation *Graphical Oracle* part indicates that the returned page is the same as the `result1.bmp`.

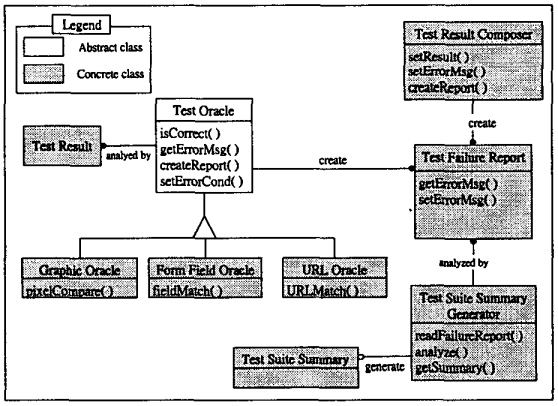

Figure 11. The Class Diagram of Test Failure Analysis Subsystem

Figure 11 shows the class diagram of TFAS. If the validation of *Test Result*, created by TES, is performed by *Test Oracle*, it will analyze the *Test Result* and create *Test Failure Report* to represent the validation result. If the validation is performed by tester, he can use *Test Report Composer* to create *Test Failure Report*. After a set of test cases are executed and validated, tester can use *Test Suite Summary Generator* to summarize test suite to see how many test cases pass the validation, which test cases fail in the validation, etc.

3.7 Test Measurement Subsystem, TMES

TMES includes test coverage measurement and analysis. Test Coverage Analyzer is designed to measure whether and how much of a test criterion is adequately satisfied. In traditional software white-box testing, programs are modeled to control or data flow, and the coverage is the percentage of statements that a set of test cases is covered if the all-statements criterion is used. In [3], Web pages or programming modules are modeled as statements, and hyperlinks are modeled as execution flow in traditional software testing. The coverage criterion in conventional software testing can also be applied here. However, As the evolution of Web application development techniques, one Web page in browser side may represent one programming module in server side. There also contains a control flow in one programming module, and the definition of test coverage mentioned above can not be applied in this situation.

4. Applying the Architecture

Based on this architecture, we implemented a prototype of Web application testing environment to demonstrate its practicality discussed in section 2.4. Figure 12 is an overview of the Web application testing environment. Tools inside the tool set are controlled through the WWW control interface, to which authorized Web browsers can connect via HTTP.

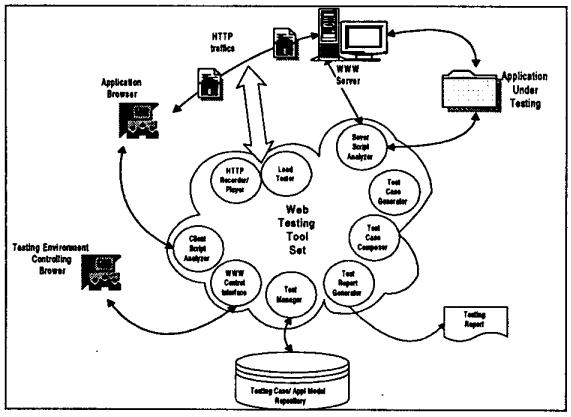

Figure 12. The Web Application Testing Environment

TCR is used to create test cases by testers semi-automatically. Before starting TCR, testers should specify the port of TCR and the name of the created test case. After pressing the button OK, the TCR is ready to record. Then, testers set the proxy property of Web browser to the TCR, and activate the Web application with Web browser. The execution scenario and input data are recorded by TCR. After pressing button Finishing Recording, TCR will store recorded data as test cases.

TCC is an editor to refine existing test cases, and creates test cases manually. To specify test cases in a more flexible way, testers can resort to handcrafting test scripts by themselves. Because the testing environment stores the outcome of test case recorders and test case generators in the test repository in the form of test scripts, test designers can change test cases by modifying test scripts using TCC.

In test cases execution, the *Test Case Executor* interprets designated test scripts, and sends corresponding HTTP requests according to the content of test script. The test results of test execution are stored in the testing log of *Test Suite/Case Repository*, from which the *Test Report Composer* summarizes testing reports.

5. Conclusion and Future Work

In this paper, we proposed a reusable architecture to construct Web application testing environments by extending a well-evaluated architecture and applying some design patterns. The architecture contains six subsystems, and the testing processes (e.g. test case generation) can be achieved with the cooperation of these subsystems. To demonstrate the usability of this architecture, a prototyped Web application testing environment is built.

Although many facets in Web applications are discussed, there are still some facets which is popular in Web application development but not mentioned. For example, many Web applications such as online ordering system are associated with a database, and many users' behaviors will cause the database accesses. We are now focusing on these issues and propose a set of components to handle the interactions between Web server and database.

Reference:

[1] Charles W. Krueger, *"Software Reuse"*, ACM Computer Surveys, page 131-183, June 1992.

[2] Debra J. Richardson, *"TAOS: Testing with Analysis and Oracle Support"*, International Symposium on Software Testing and Analysis, page 138-153, March 1994.

[3] Ji-Tzay Yang, Jiun-Long Huang, and Feng-Jian Wang, *"A Tool Set to Support Web Application Testing"*, International Computer Symposium, December 1998.

[4] Chia-Lin Hsu and Feng-Jian Wang, *"A Web Database Application Model for Software Maintenance"*, National Chiao-Tung Univertisy, Master Thesie, 1998.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison-Wesley, 1994.

[6] Mercury Interactive Corp., *"Visual Web Site Management – Mecury Interactives's Astra SiteManager"*, in http://www.merc-int.com/products/astrasmguide.html.

[7] Nancy S. Eickelmann and Debra J. Richardson, *"An Evaluation of Software Test Environment Architectures"*, International Conference on Software Engineering, page 353-364, March 1996.

[8] Rational Software, *"Visual Test 4.0 White Paper"*, in http://www.rational.com/products/visual_test/prodinfo/whitepapers/dynamic.jtmpl?doc_key=100464

[9] Reboert Martin, Dirk Riehle, and Frank Buschmann, *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.

[10] Sun Microsystems, *"SunTest Suite"*, in http://www.sun.com/suntest.

[11] Softbridge Inc., *"Web Analyst"*, in http://www.softbridge.com.

[12] Thomas Ostrand, Aaron Anodide, Herbert Foster, and Tarak Goradia, *"A Visual Test Development Environment for GUI Systems"*, International Symposium on Software Testing and Analysis, page 82-92, March 1998.