Université d'Ottawa · University of Ottawa

Université d'Ottawa • University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES
ET POSTDOCTORALES

FACULTY OF GRADUATE AND
POSTDOCTORAL STUDIES

Wei XU
AUTEUR DE LA THÈSE - AUTHOR OF THESIS

Master of Computer Science
GRADE - DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

FTEXT - Functional Testing of E-Commerce Systems with Extreme Programming and TTCN-3

R. Probert
DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

CO-DIRECTEUR DE LA THÈSE - THESIS CO-SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

L. Peyton

M. Weiss

J.-M. De Koninck, Ph.D.
LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

FTEXT - Functional Testing of E-Commerce Systems with Extreme Programming and TTCN-3

By

Wei Xu

A thesis submitted to the Faculty of Graduate and Postdoctoral Studies in partial fulfilment of the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Information Technology and Engineering

University of Ottawa

Ottawa, Ontario, Canada

July 2004

Your file Votre référence
ISBN: 0-494-01650-7
Our file Notre référence
ISBN: 0-494-01650-7

Abstract

Today's e-commerce systems have evolved from traditional middleware technologies such as CORBA and DCOM to a multi-tiered infrastructure. Java 2 Enterprise Edition (J2EE) and Microsoft .NET are two main competitors building on such architecture. Usually a multi-tiered architecture involves distinct architecture layers, components, distributed applications and enterprise resources. This complexity poses considerable risks to e-commerce systems building on such infrastructure.

To address the current problems of high testing cost and non-formal test specification, we propose a new process called FTEXT (Functional Testing of E-commerce systems with eXtreme programming and TTCN-3). FTEXT combines an agile software development process, cost-effective test tools and a formal test specification language that serve a common purpose and work together to support a cost-effective approach to test J2EE based e-commerce systems. To facilitate TTCN-3 test execution, we have designed and implemented a parser that translates TTCN-3 test script to Java source code that conforms to the selected open source test tool. The source code is then compiled into executable test code.

In the context of J2EE based system development and testing, this thesis offers the following contributions:

- Methodology: Proposed a new cost-effective model (FTEXT) for testing J2EE based e-commerce systems.
- Parser: Designed and implemented a parser to translate TTCN-3 test script and generate Java test code that conforms to the selected open source test tool for functional testing of Web-based applications.
- Case Study: Implemented our approach in a case study that tests a typical J2EE e-commerce application. The case study demonstrates the working mechanism of the FTEXT method, relevant open source tools (e.g. HttpUnit and Apache Ant) and the capability of our TTCN-3 to Java parser.

Acknowledgements

I would like to acknowledge the help that I have received during this research.

I am indebted for my supervisor, Professor Robert Probert, for his continuous support, his invaluable guidance, patience and intelligent comments.

Grateful thanks to:

- Professor Michael Weiss and Professor Liam Peyton for kindly reviewing this thesis and accepting to be on my defence committee.
- Dr. Ostap Monkewich, Dr. Alan Williams, Dr. Daniel Amyot and Mrs. Louise Desrochers for their unconditional help and resourceful advice.
- The ASERT group in the University of Ottawa for their help, comments, and valuable discussions.
- Communications and Information Technology Ontario (CITO), the Natural Science and Engineering Research Council (NSERC) for their financial support.
- My friends for their consideration and encouragement.
- My family for their endless support to me.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1 Introduction

1.1 Background and Motivation

Today's e-commerce systems have evolved from traditional middleware technologies such as CORBA and DCOM to a multi-tiered infrastructure. Java 2 Enterprise Edition (J2EE) and Microsoft .NET are two main competitors built on such an architecture. Usually a multi-tiered architecture involves distinct architecture layers, components, distributed applications and enterprise resources. This complexity poses considerable risks to e-commerce systems built on such infrastructures [Johnson03].

Testing is a critical issue that can make a difference between success and failure in e-commerce systems both from business and technology perspectives. Due to its inherent complexity, J2EE e-commerce application testing needs to apply a wide range of testing techniques and tools. Commercial test tools are often expensive and have a steep learning curve. One the other hand, open source tools are becoming more prevalent over the past few years thanks largely to a lightweight programming methodology called eXtreme Programming (XP) [Beck99], which elevates testing to the key activity of the software development process. The combination of open source tools and XP offers a cost-effective means for rapid development of large, complex and distributed enterprise systems. Using formal methods for testing and an international standard test specification language, TTCN-3, in our opinion would contribute to improvements to software quality, especially for specifying and testing complex and distributed systems. The Test and Test Control Notation version 3 (TTCN-3) [ETSTI01] has been shown to be effective for software development in the telecommunication and data communication field [Schieferdecker01a, Schulz02, Wiles02, Schieferdecker03]. However, to our knowledge, little similar research has been done on formal specification or testing of J2EE-based systems in TTCN-3.

Our motivations are the rising cost of testing and insufficient test quality. Our approach is focused on how to improve test quality and decrease test costs.

To improve test quality, in this thesis, we proposed:

- Introducing a formal international standard test specification language, TTCN-3
- Using open source tools that facilitate automatic and continuous testing (a fundamental part of XP development)
- Using a parser we developed to translate TTCN-3 script to Java code which is applicable to selected open source tools
- Integrating best web testing practices and TTCN-3 test case design

To decrease test costs we proposed:

- Adopting open source tools
- Using our tools and methods which support development based on the J2EE architecture, which is a widely accepted enterprise application development platform

1.2 Problem Statement

Our goal in this thesis is to provide a cost-effective approach:

- To test J2EE web-based systems such as e-commerce systems efficiently (to lower cost)
- To make the test process more systematic and formal (to improve test effectiveness and test quality)

1.3 Approach of Thesis

To address the current problems of high testing cost and non-formal test specification, we propose a new process called FTEXT (Functional Testing of E-commerce systems with eXtreme programming and TTCN-3). FTEXT combines an agile software development process, cost-effective (open source) test tools and a formal test specification language (TTCN-3) that serve a common purpose and work together to support a cost-effective approach to testing J2EE based e-commerce systems. To facilitate TTCN-3 test

execution, we have designed and implemented a parser that translates TTCN-3 test script to Java source code that conforms to the open source test tool. The steps in the testing process are:

1. Apply best web testing practices to design and specify test cases in TTCN-3 for e-commerce systems functional testing
2. Select an appropriate open source functional test tool
3. Design and implement a parser that translates the TTCN-3 test scripts to Java test code that conforms to the input constraints for the selected open source test tool
4. Introduce new requirements on the fly and modify test cases according to new requirements Apply extreme programming, open source tools and best web test practices to Case Study consisting of an e-commerce system as SUT (System Under Test) and utilize a simulated e-commerce client to stimulate the SUT
5. Repeat cycle until adequate functional test coverage is achieved

Our approach in this thesis is as follows:

1. Apply extreme programming, open source tools, selected best web test practices, and develop a parser to translate appropriately from TTCN-3 to Java code to a well-known case study consisting of an e-commerce system as SUT (System Under Test) and utilize a simulated e-commerce client to exercise the SUT
2. Introduce new requirements on the fly and show how to modify test cases according to new requirements
3. Observe and analyze test results
4. Assess our testing approach and give directions for future work

1.4 Contributions of Thesis

In the context of J2EE-based system development and testing, this thesis offers the following contributions:

- Methodology: Proposed a new cost-effective approach (FTEXT) for testing J2EE based e-commerce systems. The approach combines a formal test specification

language, an agile development and test methodology, open source test tools and Web functional testing best practices.

- Parser: Designed and implemented a parser to translate TTCN-3 test scripts and generate Java test code that conforms to the selected open source test tool for functional testing of Web-base applications.
- Case Study: Applied our approach to a well-known case study that tests a typical J2EE e-commerce application. The case study is a practical showcase that demonstrates the working mechanism of the FTEXT approach, relevant open source tools (e.g. HttpUnit and Apache Ant) and the capability of our TTCN-3 to Java parser.

1.5 Thesis Organization

The thesis is organized as follows:

Chapter 2 gives a brief background of software development processes and basic software testing techniques.

Chapter 3 introduces the middleware architecture and testing issues and strategies for testing e-commerce systems, followed by descriptions of web functional testing best practices that are used in our test approach.

Chapter 4 is a brief introduction to the international standard testing language TTCN-3, its history, evolution and some main concepts. An example TTCN-3 application at the end of this chapter gives the flavor of using TTCN-3 for formal testing.

Chapter 5 overviews briefly the extreme programming methodology and some of its principales and rules. Some open source tools (JUnit, Cactus, JUnitPerf and Apache Ant) that support XP are introduced. At the end of this chapter we present the main part of the thesis - our cost-effective testing approach in detail. We also explain the design and implementation of our TTCN-3 to Java parser.

Chapter 6 uses a case study to demonstrate and evaluate our test approach described in chapter 5. A detailed description of the SUT, the TTCN-3 testing specifications and the parsed Java test code along with configuration and execution steps of these tests are explained. The obtained test results and the analysis based on test results are explained in detail. We also make some changes to the original test scripts in TTCN-3 on the fly and replay the whole test process to show the feasibility and flexibility of the parser.

Chapter 7 summarizes the pros and cons of our test approach and gives some recommendations for future work.

Chapter 2 Software Testing Fundamentals

This chapter covers broad testing topics in general so as to give the background knowledge for consecutive chapters. The introduction of software development process models leads to extreme programming which is the software methodology applied in this thesis, some of the main aspects of software testing strategies and relevant techniques will be presented followed by the introduction of software test automation along with some of the automation test tools including WebLoad [WebLoad], and E-valid [evalid].

2.1 Software Development Process Models

Developing high quality software systems is a complex and time consuming process. In order to control this process, reduce the complexity and uncertainties of building such software systems, developers need to follow some kind of framework that introduces certain steps and rules during the overall development process.

Software development models are the frameworks that describe how people should go about developing software systems. These frameworks define different phases of the development process, such as planning, requirements analysis, design, testing and maintenance.

Over the years, many organizations have slowly become aware of the importance of a well defined and well documented software development model to the success of their software projects and various models have emerged to support the development of high-quality software products. We describe some of the typical methods from a software development historical viewpoint.

2.1.1 Big Bang Model

The Big Bang Model actually should not be called a software development model. It just gives the scenarios at the time when people develop software without an idea of software development models in mind. In the old days, investment, energy and time poured in to develop software and write code without any planning, scheduling or documentation. Testing is performed after the product is build and ready to deploy. Software products build in this fashion bring about tremendous problems even before it is shipped to the customer.

Without scheduling and planning, there is no means to measure the progress and quality of the project, which means customers don't have the confidence that the software will be delivered in time and perform the expected functionality. As there is no detailed documentation of requirement specifications and analysis, it is hard to trace bugs in such products, and as time goes on, more bugs will be found and this will further aggravate the situation.

Though we did not learn any thing on how to build good software products from this model, it really set a reversed example for us to avoid building such anguish products.

2.1.2 Code and Fix Model

Code and Fix is the prototype of a formal software development model. With a general idea of what to build in mind and some simple design, developers start working on a code-and-fix circle to develop the software. Testers in the development team will run tests for developed functions or features and report bugs to developers who fix the bug and make a new release back to the tester.

Like the Big-Bang model, the code and fix development does not have a specific indication on when to release the product. It will be in a constant cycling as long as conditions permit.

Though not much planning or scheduling is performed ahead of the project, it is not a bad idea to apply such a model for small projects like developing prototypes or demos, as the development team can show their results quickly and it is reported that the code-and-fix model had been used on many large projects.

2.1.3 Waterfall Model

The Waterfall model is considered to be one of the earliest formal software development models. In such a model, activities are divided into different stages and carried out in order. Figure 2.1 shows different stages of the Waterfall model. Each stage has a detailed specification and the result of one stage will act as the starting point for the next stage, for example, the design stage will depend on the result of the analysis stage. At the end of each stage, there will be a review to verify if the work of this stage is fulfilled before the project goes to the next stage.

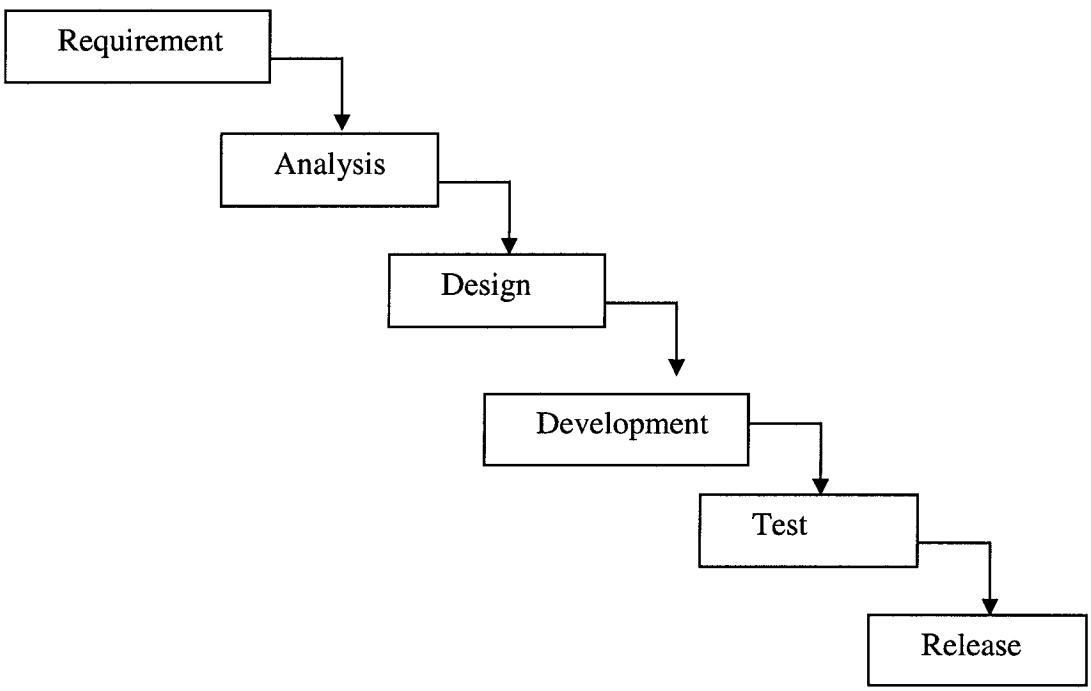

Figure 2.1 Waterfall Model

However, this model rarely fits into the real world as it assumes that system requirements are fixed and do not change throughout the development process, but in reality, neither developers nor users could possibly have a precise and accurate understanding of the whole system at the beginning of the project. To accommodate this situation, another approach called the Spiral Model comes into play.

2.1.4 Spiral Model

Instead of specifying detailed definitions of the entire system at first, the Spiral Model starts on a very small scale of the project. It works in an iterative process intended to help manage risks. Each iteration process involves planning, analysis, design, implementation and evaluation. Developers only define the highest priority features, implement these features, get feedbacks from users or customers, then go back to define and implement more features in smaller chunks.

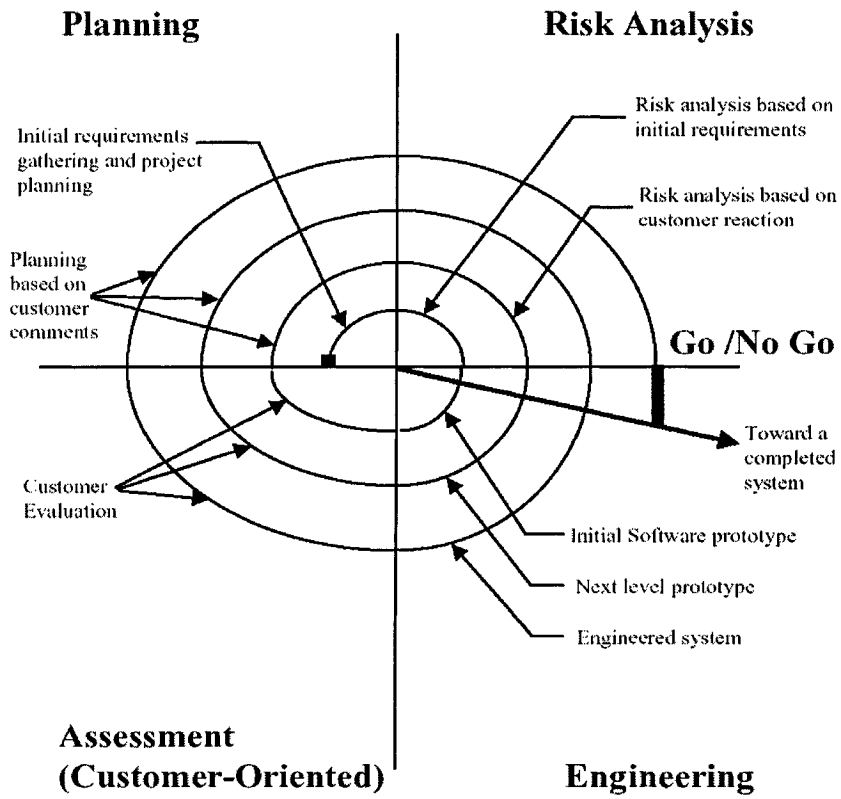

Figure 2.2 Spiral Model

The advantage of this model is that it is a risk-oriented process, that is with every iteration costs increase but risks decrease. The disadvantage is that it is complicated, and requires attentive management.

2.1.5 Rapid Application Development (RAD)

Through the years of software development practice, people realize that it is unrealistic to expect the development teams to perform a complex task correctly at the first time. However, they are extremely good at making an adequate beginning and then making many small refinements and improvements. Rapid Application Development model is such an approach that is more appropriate for human nature.

RAD projects are typically comprised of small teams including developers, end users, and IT technical resources and work in short, iterative development cycles. Iteration allows for effectiveness and optimizes speed, informal communication and simple project management.

2.1.6 Rational Unified Process (RUP)

RUP is the result of many years of software development's best practices that is suitable for a wide range of projects and organizations. These best practices [Kruchten00] including:

- Develop software iteratively.
- Manage requirements.
- Use component-based architectures.
- Visually model software.
- Continuously verify software quality.
- Control changes to software.

Rational Unified Process assigns tasks and responsibilities within a development organization. Its goal is to ensure the production of high quality software that meets the needs of its end users within a predictable schedule and budget.

RUP can be described in terms of two dimensions: time and content. Figure 2.3 provides a graphical representation of these dimensions [Jacobson99].

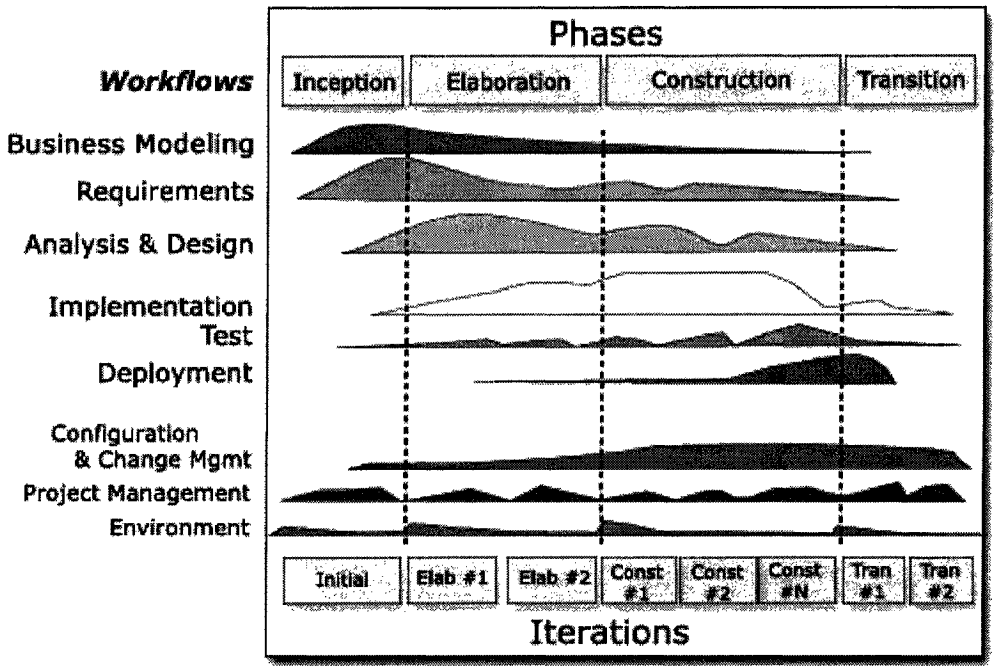

Figure 2.3 Time and Content Detentions of RUP

The horizontal dimension represents time and shows the lifecycle aspects of the process. The vertical dimension represents core process disciplines (or workflows), which logically group software engineering activities. As we can see the "humps" in each phase is different, representing each phase emphasis on different activities, for example, in the beginning inception phase, more time is spent on business modeling and requirement analysis, while in the construction phase, implementation, deployment and configuration should be the main work load.

2.1.7 Extreme Programming (XP)

Agile programming is becoming popular recently [Fowler00]. XP is a distinguished example of such approaches. Extreme programming is a lightweight software

development methodology that focuses on coding as the main task. It contains 12 core practices to realize the four values that contribute to the quality of software development.

Testing is one of these core XP practices. Programmers are expected to write unit and functional tests code even before the application is developed. Production code is developed trying to pass the test that is prepared in advance. Customers are expected to be an integral part of the project team and to help develop scenarios for acceptance testing. Acceptance tests are preferably automated, and are modified and rerun for each of the frequent development iterations. A detailed description of XP is in chapter 5.

2.2 Software Quality Problems and Causes

2.2.1 Real World Failures Caused by Poor Software Quality

In October of 1999 the $125 million NASA Mars Climate Orbiter spacecraft was believed to be lost in space due to a simple data conversion error [Cowen99]. It was determined that spacecraft software used certain data in English units that should have been in metric units. Among other tasks, the orbiter was to serve as a communications relay for the Mars Polar Lander mission, which failed for unknown reasons in December 1999. Several investigating panels were convened to determine the process failures that allowed the error to go undetected.

In February of 2003 revealed that the U.S. Treasury Department mailed 50,000 Social Security checks without any beneficiary names. A spokesperson indicated that the missing names were due to an error in a software change. Replacement checks were subsequently mailed out with the problem corrected, and recipients were then able to cash their Social Security checks.

In March of 2002 it was reported that software bugs in Britain's national tax system resulted in more than 100,000 erroneous tax overcharges. The problem was partly attributed to the difficulty of testing the integration of multiple systems.

In April of 1999 a software bug caused the failure of a $1.2 billion U.S. military satellite launch, the costliest unmanned accident in the history of Cape Canaveral launches. The failure was the latest in a string of launch failures, triggering a complete military and industry review of U.S. space launch programs, including software integration and testing processes. Congressional oversight hearings were requested.

2.2.2 Reasons for Software Errors [Hower]

Miscommunication or no communication

This could happen between developers and between developers and customers regarding to specify what an application should or shouldn't do.

Software complexity

The complexity of current software applications can be difficult to comprehend for anyone without experience in today's software development. New technologies, applications, architectures and frameworks and methods all contributed to the exponential growth in software and system complexity.

Changing requirements

There are dependencies among parts of the project that is already completed, any change may affect the known and unknown dependencies and likely to cause problems.

Time pressures

Scheduling of software a project is difficult and often requiring a lot of guesswork. People are likely to make mistakes when they are trying to meet unrealistic deadlines.

Poorly documented code

Usually, programmers don't like to document their code. This will bring the problem of maintaining and modifying code that is badly written or poorly documented which could

Software development tools

Visual tools, class libraries, compilers, scripting tools, etc. often introduce their own bugs or are poorly documented, resulting in added bugs.

2.3 Software Test Design Strategies and Methodologies

There are various methods for software testing. Traditionally, they are split into three main categories: Black Box, White Box and Grey Box testing.

2.3.1 Black Box Test

Black box testing is also known as Behavior or Functional testing. IEEE defines Black box testing as [IEEE90]: Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions. Black-box test design is usually described as focusing on testing functional requirements.

In Black box testing, testers treat the entity being tested as a black box, they don't have exposure to program code and they don't need to care about the internal working mechanisms of the system, but only its input-output relationship that is described in system specification.

The following techniques fit into Black box testing. They provide guidelines for test case design and test data selection practices to ensure a certain amount of quality in the test cases they generate.

Equivalence Partitioning

Equivalence Partitioning [Myers79] helps reduce the size of the domain of each black box, it suggests the construction of test cases by first splitting the domain into equivalence partitions or classes. These equivalence classes are classes of input data that, according to the specification, are treated identically. For example, if the specification contains something like the following example:

```
if x<100 then
<something>
else
```

```
<something else>
end if
```

Then the values of x<100 would be one equivalence class and those values of x>=100 would belong to another equivalence class. This test method suggests that constructing test cases with one value from each class is sufficient. The test set should also contain values that are invalid or would be expected to cause errors.

Boundary-Value Analysis

Boundary-value analysis [Roper94] is an extension of equivalence partitioning. The idea is to focus on the parts of the code that is more prone to errors (on-by-one, off-by-one). In the above example, the partition boundary is x=100, we would concentrate the test cases around the values of x=100, 99, and 101.

Cause-effect Graphing

Cause-effect graphing is a technique where by the relationship between "effects" and their "causes" is explored to aid the construction of a test set. A cause is usually something that will cause a response, an input or some other stimulus. An effect is an observable response, like an output or a change in state. The idea is that, first, all the causes and their effects on the system are listed; then, create a graph to illustrate the relationship between the different causes and effects. This graph is then turned into a decision table that shows what effects are caused by every possible combination of causes (inputs). The final step is to convert this decision table into a series of test cases.

This method provides more guidance than the previous two because it forces the generation of test cases that will exercise combinations of inputs. The expected outputs are also generated as part of the test set generation process. The problem is that the construction of the graph quickly becomes very complex as the number of causes and effects increase.

2.3.2 White Box Test

White box test is also called Structural test that allows the use of the implementation to construct the test cases. This makes it possible to construct nearly exhaustive test cases, unusual or exceptional cases should be constructed to look for errors around these points. The following techniques are used for White box testing:

Statement Coverage Testing

Statement coverage testing aims to execute every state in the source code at least once. Test cases are generated to exam the source code, ensuring all routes through the algorithm are evaluated. The problem with this method is that, simply because every line of code is executed does not mean that all the paths through the algorithm will be tested (path coverage).

Branch Coverage Testing

Branch coverage testing cause the construction of test cases that execute the true and false parts of all the branch statements in the code (if and case). Although branch coverage is an improvement upon simple statement coverage there are still faults that this method may not find. For example when you have complex conditionals like (a=1 AND b<2) OR (c>100), branch cover may not be able to tell the difference between this and (a=1 AND b<2) OR (c>=100).

Condition/Decision Coverage

This technique examines each of the conditionals identified by branch cover. Each of the sub-conditionals (a=1 is a sub-conditional of (a=1 AND b<2)) of a conditional is evaluated as true and false. In addition to the requirement of branch cover, the true and false branches of each branch must be evaluated. This technique solves the shortcoming of Branch cover in that it will find the difference between conditionals like (a=1 AND b<2) OR c>100 and (a=1 AND b<2) OR c>=100.

2.3.3 Grey Box Test

Grey box test is based on partial knowledge of the internals of the system under test (SUT). This is not to be confused with white box test, which attempts to cover the internals of the SUT in detail. In the Grey box mode, testing is carried out from the outside of the SUT just as it with Black box, but test case generation and selection are informed by part of the knowledge of how the underlying components operate and interact.

Grey box test is especially important with Web and Internet applications, because the Internet is built around loosely integrated components that connect via relatively well-defined interfaces.

2.3.4 Summary of Software Test Design Methodologies

Black box and white box are two of the most used test design strategies. Black box test design is usually described as focusing on testing functional requirements. White box test design allows one to look inside the "box", and it focuses specifically on using internal knowledge of the software to guide the selection of test data. Grey box test design takes advantages of black box and white box models and it fits more into real world testing requirements.

It is important to understand that these models are used during the test design phase, at the test implementation phase, any level of testing (unit testing, system testing, etc.) can use any test design methods. Unit testing is usually associated with structural test design, but this is because testers usually don't have well-defined requirements at the unit level to validate. In practice, it hasn't proven useful to use a single test design method. Testers have to use a mixture of different methods so that they take advantages of these test strategies and avoid drawbacks and limitations of these techniques.

2.4 Test Automation and Test Tools

Testing is usually not sufficient compared to other activities within the software development process, but with today's reduced development cycle time and more

complex systems, testing becomes even more difficult. Software developers are facing the problem of keeping up with the explosive pace of software development while at the same time retaining satisfactory test coverage and reducing risk. Test automation may be the answer to this problem.

2.4.1 Benefits of Test Automation

The benefits of test automation include:

- Reduce testing time. For a sophisticated product, manual testing may require many people working for months, but the same testing task may achieved within hours in automatic testing.
- Consistent testing procedures. With a complex testing process, manual testing often introduces inconsistent into test process humans make when they get tired after multiple repetitions. An automated test suite will avoid the errors and make sure the exact testing process is performed repeatedly.
- Reduce QA costs. Automated testing has an extra cost to develop, but over multiple product releases with multiple cycles per release, this cost is quickly reduced.
- Improve testing productivity. With its much shorter execution time an automated test suite can be run multiple times over the course of a product development cycle.
- Improve product quality. Automated testing detects functional and performance issues more efficiently.

2.4.2 Test Automation Applications

Test automation parallels software development that moves from desktop applications, to client/server applications, and to today's Web-enabled applications. We describe test automation from this application evolution perspective:

Desktop Applications

Software development automation for desktop applications focused around improving the time it took to code and build software. Desktop application frameworks give software developers a quick way to start coding. The frameworks provide the library calls to

initialize the operating system, construct the application's initial windows and menus, and handle functions common to many applications (for example, file management, printing, and memory management.) Test automation focused on improving the time it took to test a desktop application for functionality.

Client/Server Applications

Client/server applications are very popular with businesses because business logic are centralized on the server makes system maintenance, deployment and development much easier. Client/server application test automation provides the functionality of desktop application test automation plus these:

- Client/server applications operate in a network environment. The tests need to not only check for the function of an application, they need to tests how the application handles slow or intermittent network performance.
- Automated tests could be performed to determine the number of client applications a server is able to efficiently handle at any given time.
- The server is usually a middle tier between the client-application and several data sources. Automated tests need to check the server for correct functionality while it communicates with the data sources.

Web-enabled Applications

Web-enabled applications offer the advantages of client/server applications: efficient system maintenance, deployment and development, plus they reduce the burden of client-side software maintenance, increase the flexibility of data center architecture to support multiple, diverse servers. An ideal web-application test tool should offer these features:

- A friendly, graphical user interface to integrate the record, edit and run-time script functions.
- A recorder that watches how an application is used and writes a test script for you.
- A playback utility that drives a Web-enabled application by processing the test script and logging. The playback utility also provides the facility to playback several concurrently running copies of the same script to check the system for scalability and load testing.

- A report utility to show how the playback differed from the original recording.

Web-enabled application test tools are very popular in today's IT industry. Some commercial tools like Mercury Interactive LoadRunner [Mercury], Segue SilkPerformer [Segue], RadView WebLoad [RadView], and Empirix eLoad [Empirix] or open-source tools such as PushToTest TestMaker [PushToTest], Apache JUnit [JUnit] and JMeter are available on the market. All these tools share some common features like graphical interface, recorder, playback, and report utility.

2.4.3 Automatic Testing Tools

Automatic testing tools could be used at all phases of development. We need tools that are practical, have short learning curves, and could improve software development productivity. The following test tools are mainly used in testing web applications.

eValid

eValid from Software Research, Inc is a tool for functional, performance, and load testing of Web applications. It uses the same Web cache, history, cookies, plug-ins, and other objects as the installed version of Microsoft Internet Explorer. Its ease of use is due to its familiar browser interface (see Figure 2.4). eValid testing features are included in the browser itself. With eValid, testers can create a more realistic simulation that generates HTTP requests without composing and rendering the results.

Figure 2.4 eValid User Interface

Functional testing is accomplished with eValid test scripts (similar to JavaScript). Scripts simulate user sessions, and allow test validation of any or all links, text, images, or other objects encountered in the Web application.

Performance testing is accomplished automatically by eValid whenever a script is executed. The tool keeps track of timing information, logging time to download entire pages and all individual components. Testers can view performance results in graphical or spreadsheet formats, and use multiple playbacks to study how script execution times vary.

Load testing is accomplished with special load testing scripts. A load testing script allows the tester to run multiple concurrent eValid test scripts, each in a different browser window. Each line of a load test script spawns an eValid session that repeatedly runs a test script.

WebLoad

RadView's WebLoad is designed to provide developers and testers with Web application scalability information through a comprehensive set of diagnostic tools that attempt to

stress test a Web application in the areas of performance and robustness. WebLoad generates a load of virtual clients that simulate browsers, clients or HTTP requests. These virtual clients act as real-world traffic and stresses to the application being tested (ABT). WebLoad executes a number of predefined test scripts that can be created by developers or administrators using a JavaScript-based language. WebLoad monitors the performance response of the Web application as it executes these test scripts.

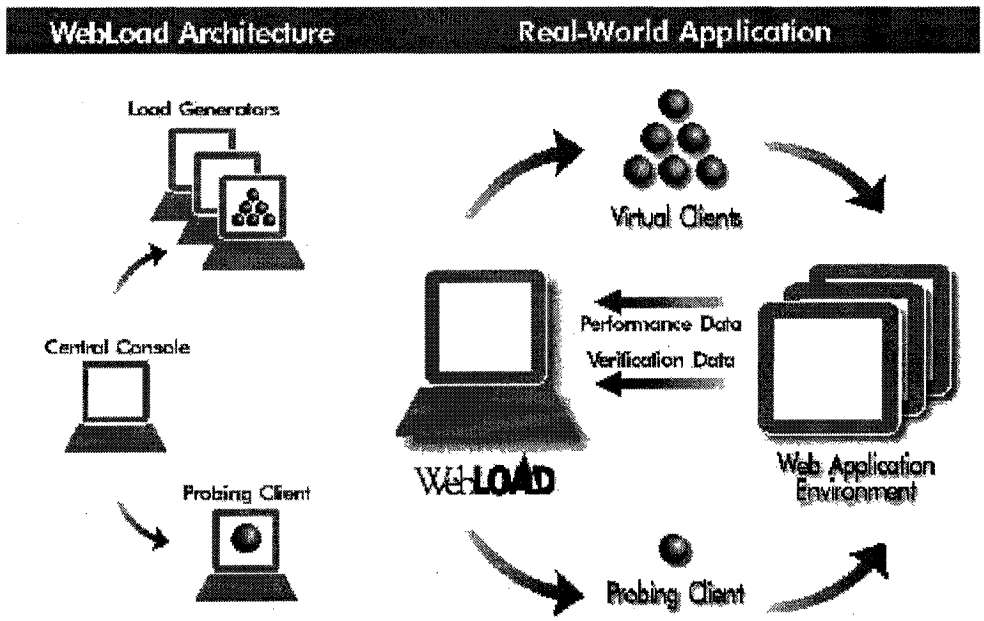

Figure 2.5 WebLoad Configurations

When testing an application, WebLoad can also run a probing client, which is a single virtual user for whom WebLoad collects more precise measurements of the application's responses. The probing client gives deeper insight into the application's behavior under the stress of the load applied by the load generators. Figure 2.5 shows the configuration of load generators and probing client.

The ABT cannot distinguish a virtual client from a real one. WebLoad measures the time the ABT takes to send back a response and records this information, displaying it graphically on the console (see Figure 2.6).

WebLoad uses the JavaScript language to write the agendas, which virtual clients run. These agendas use the Document Object Model (DOM) to interact with web pages, dynamic HTML, and nested links. Agendas simulate a user's behavior, down to the sleep time or thinking time between a live user's actions.

It is not necessary to know JavaScript or the DOM to create an agenda. WebLoad includes the Agenda Authoring Tool (AAT) for automated creation of agendas. The AAT opens a web browser and records the user's activities in an agenda until he stops the recording.

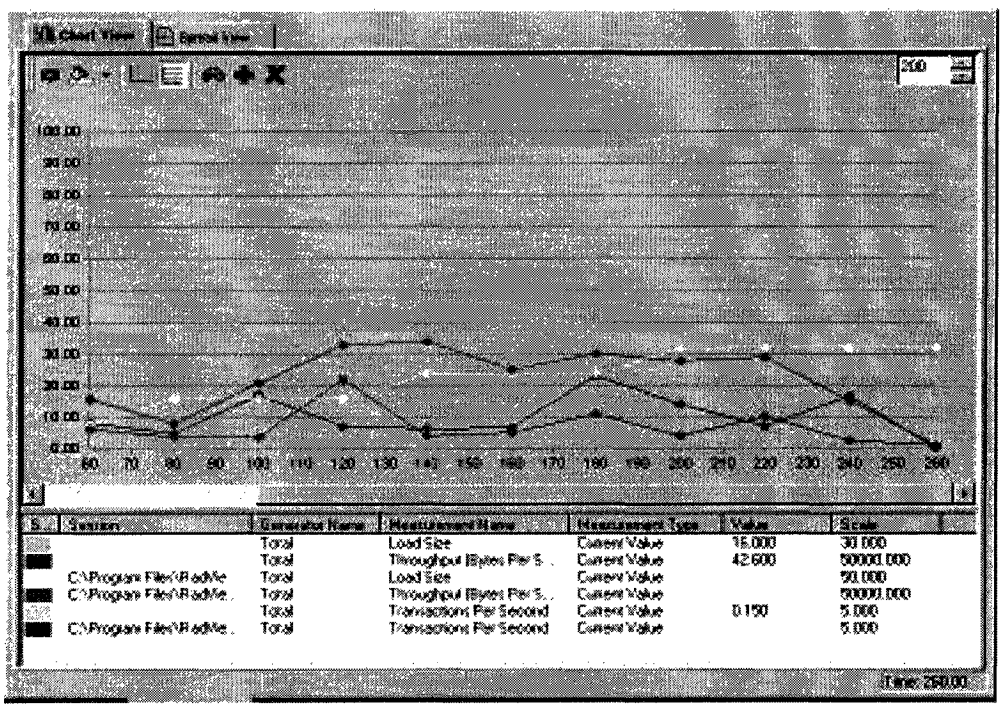

Figure 2.6 WebLoad Graphical Performance Reports

WebLoad provides more flexible reporting controls. Administrators can define specialized metrics or timers for single clients. In addition, administrators or developers can generate detailed status and load reports, and view test reports both in real-time and at the conclusion of a WebLoad session.

Chapter 3 E-commerce Middleware Architecture and Functional Test Strategy

In this chapter, we introduce e-commerce system middleware architectures especially J2EE which will be the system under test in our case study in chapter 6. In the subsequent sections we focus on testing issues in term of e-commerce applications and explain best practice strategies for e-commerce functional testing.

3.1 E-commerce Middleware Architecture

3.1.1 A Brief History of E-commerce

Electronic commerce was first developed in the early 1970s with innovations such as electronic funds transfers, in which funds could be transferred electronically. However, the extent of the applications was limited to large corporations and financial institutions. Electronic Data Interchange (EDI), a technology used to electronically transfer documents such as purchase orders, invoices, and electronic payments between firms doing business. This new application enlarged the participation group from financial institutions to manufacturers, retailers, services and many other types of business. More new electronic commerce applications followed, ranging from stock trading to travel reservation systems.

As the Internet became more commercialized and users rushed in to participate in the World Wide Web in the early 1990s, the term electronic commerce appeared. Electronic commerce applications rapidly expanded. One reason for this rapid expansion was the development of new networks, protocols, software and specifications. The other reason was an increased in competition and other business pressure [Turban03]. Since 1995, Internet users have witnessed the development of many innovative applications ranging from interactive advertisement to virtual reality experiences. Almost every medium and large sized organization in the world now has Web site, and most large U.S. corporations

have comprehensive portals through which employees, business partners, and the public can access corporate information. Many of these sites contain tens of thousand of pages and links. In 1999, the emphasis of electronic commerce shifted from B2C to B2B.

E-commerce technology made dramatic progress parallel to the prevalence of e-commerce applications. Most of today's software applications are complex, distributed and object-oriented. E-commerce systems are no exception. The object-oriented design of e-commerce applications can lead to the idea of component-based architecture. A component is really a running instance of an object. Just as program is the basic entity and process is a running instance of it, an object is the basic entity that becomes a component when it has to be actually used.

Many e-commerce applications are component-based. This means that more than one component interact with another to create a full-fledged web-based application. This approach has created the new term middleware. Middleware holds all the components together. These components can be distributed. Therefore, there must be a common approach for interaction among these components. Middleware allows a distributed set of objects to interact with each other to create an e-commerce application. Actually, middleware is nothing but the glue between the client and the server. It is this approach that has led to the third tier in an e-commerce application, namely the application server.

3.1.2 Middleware Architectures for E-commerce Systems

Middleware is employed to enable the seamless integration of the applications with the various types of servers and databases. Increasingly however middleware is used in the development of applications that communicate across platforms as they traverse the Internet. Middleware usually makes implementing distributed applications easier, makes them interoperate with each other, and makes them more robust. There exist four relevant middleware architectures: CORBA, J2EE, Microsoft .NET and Web Services. CORBA has been used from the beginning of time while J2EE and Microsoft .Net could be considered the second generation of middleware infrastructure. Web Services is a relatively new concept, but has gained considerable popularity.

CORBA

CORBA is a flexible kind of middleware that is suitable for integrating legacy systems and building critical systems. The price to pay for this flexibility is that there is a steep learning curve programming with CORBA.

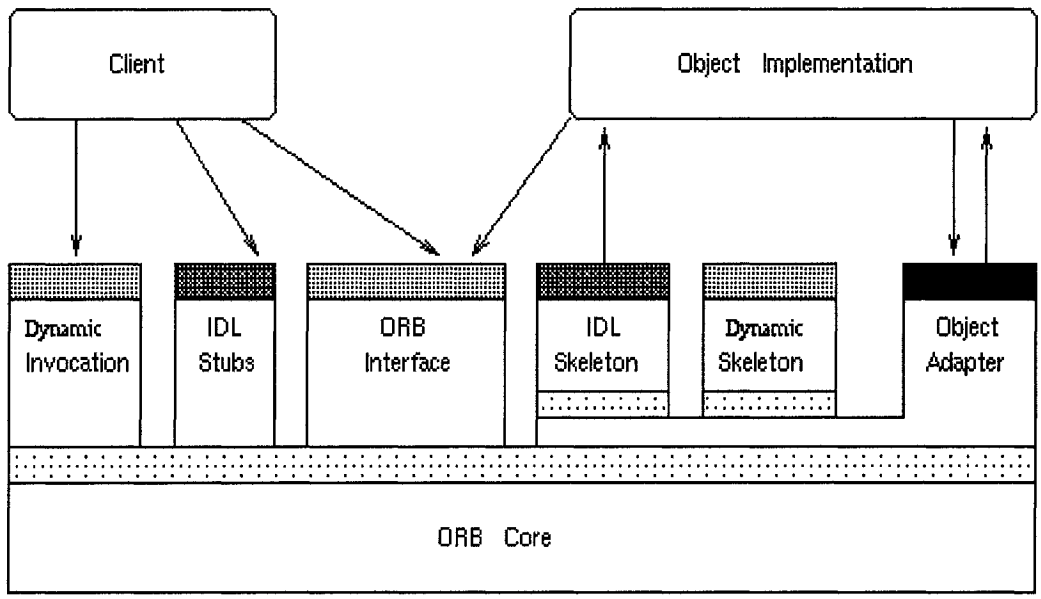

Figure 3.1 CORBA Architecture

Figure 3.1 explains the main components of the CORBA architecture. The central component of CORBA is the Object Request Broker (ORB). Its goal is to identify and locate objects, handle connection management and deliver data.

ORB provides the basic functionality of passing the requests from clients to the object implementations on the other side of the network. In order to make a request the client can communicate with the ORB through the IDL stub or through the Dynamic Invocation Interface (DII). The stub represents the mapping between the language of implementation of the client and the ORB. Thus the client can be written in any language as long as the implementation of the ORB supports this mapping. The ORB then transfers the request to

the object implementation which receives the request through either an IDL (Interface Description Language) skeleton, or a dynamic skeleton.

J2EE

The Java 2 Platform, Enterprise Edition (J2EE) is designed to simplify complex problems with the development, deployment, and management of multi-tier enterprise solutions. J2EE is an industry standard that describes agreements between applications and the containers, which is the running environment for these applications. As the initiator, Sun Microsystems collaborated with other vendors of e-Business platforms, such as BEA, IBM, and Oracle, in defining J2EE. Sun also initiated the Java Community Process (JCP) to pursue new ideas to improve J2EE over time. The J2EE camp set up a mechanism to give customers choice of vendor products and tools, and to encourage best of breed products to emerge through competition.

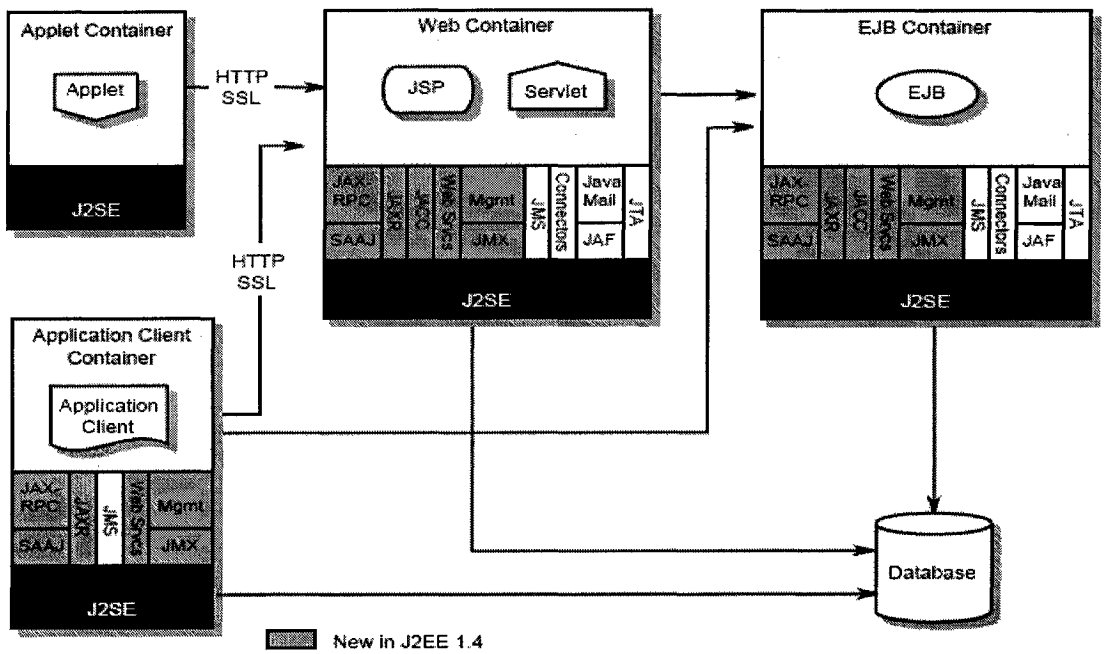

Figure 3.2 J2EE Architecture Diagram

J2EE architecture is based on the Java programming language. Developers write source code in Java, which is then compiled into "bytecode" (a cross-platform intermediary between source code and machine language). The Java Runtime Environment (JRE) interprets this "bytecode" and executes it at run-time. This is the main idea of "write once, run anywhere". Figure 3.2 depicts the component techniques in J2EE architecture [J2EE_spe]. This is a typical 3-tier application. The presentation tier includes Applet and Application client. The middle tier represents business logic facilitated by JSPs, Servlets and EJBs. The backend database tier accessible by other tiers is a separated layer in the application.

In general, two main architectures are adopted when developing J2EE applications. The first architecture utilizes Servlet and JSP in the middle tier to serve clients and process the business logic. It is also called web-centric application design as Servlets and JSPs live in the web container. This architecture is depicted in Figure 3.3. Small to medium-size applications use this design model.

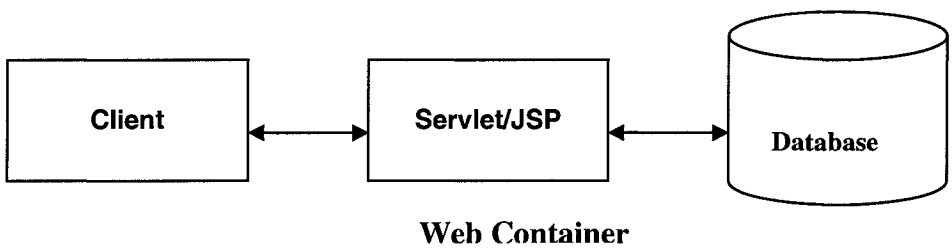

Figure 3.3 Web-centric J2EE Application Design Model

The second architecture includes the use of J2EE server and Enterprise JavaBeans (EJB) and this is especially useful for large enterprise applications that need scalability. The architecture is shown in Figure 3.4.

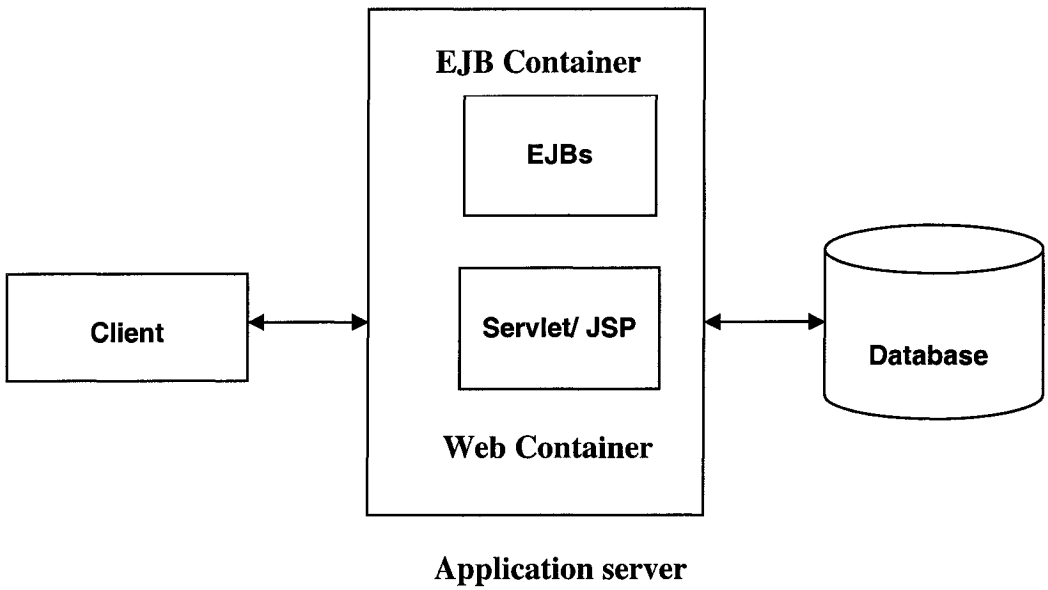

Figure 3.4 Three-tiered J2EE Application Design

Microsoft .NET

Microsoft .NET [Microsoft] is largely a rewrite of Windows DNA (Windows Distributed interNet Applications Architecture) [Blexrud00], which was Microsoft's previous platform for developing enterprise applications. Windows DNA includes many proven technologies that are in production today, including Microsoft Transaction Server (MTS) and COM+, Microsoft Message Queue (MSMQ), and the Microsoft SQL Server database.

The new .NET framework replaces these technologies, and includes a web services layer as well as improved language support. Microsoft .NET is innovative in that it doesn't compile applications into native code. Instead, the compilation is now a two-step process. The developer source code compiles into Common Intermediate Language (CIL). Then, the Common Language Runtime (CLR) compiles the code into native code at execution.

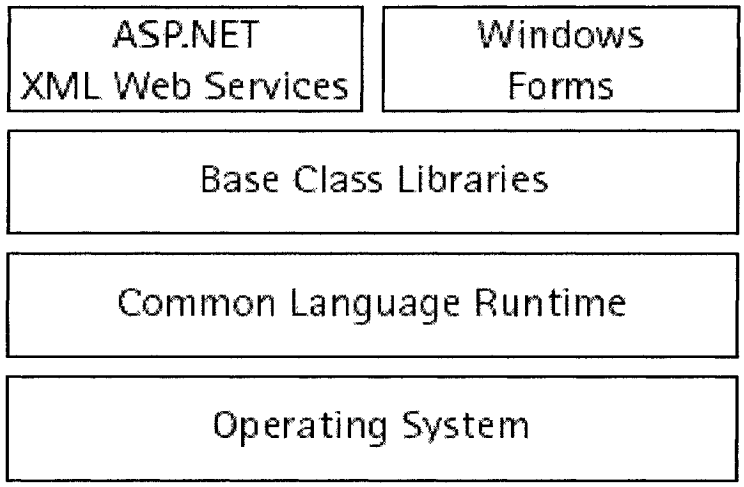

Figure 3.5 Microsoft .NET Architecture

In Figure 3.5, the layer on top of the CLR is a set of framework base classes. These classes support rudimentary input and output functionality, string manipulation, security management, network communications, thread management, text management, reflection functionality, collections functionality, as well as other functions.

On top of the framework base classes is a set of classes that extend the base classes to support data management and XML manipulation. These classes, called ADO (ActiveX Data Object), support persistent data management—data that is stored on backend databases. Alongside the data classes, the .NET Framework supports a number of classes to let you manipulate XML data and perform XML searching and XML translations.

Classes in three different technologies (including web services, Web Forms, and Windows Forms) extend the framework base classes and the data and XML classes. Web services include a number of classes that support the development of lightweight distributed components, which work even in the face of firewalls and NAT software. These components support plug-and-play across the Internet, because web services employ standard HTTP and SOAP.

Web Services

Web services enables enterprise applications to communicate with each other in a platform and programming language independent way. Even though the concept of Web services is new, the design principal from an architectural perspective is not. Web service communication is actually based on previously established middleware design principles such as RPC, RMI, DCOM, and CORBA. What distinguishes Web services from other types of Web-based applications is that operations and data exchange between Web services are described in standardized XML (as opposed to a proprietary binary standard) and supported globally by most major technology firms.

The underlying technologies of the Web services including three W3C standards: SOAP (Simple Object Access Protocol), XML, and WSDL (Web Services Description Language). Other technologies include UDDI (Universal Description, Discovery, and Integration) and of course, HTTP.

SOAP provides a standard packaging structure for transporting XML documents over standard Internet protocols, such as SMTP, HTTP, and FTP. By having a standard transport mechanism, heterogeneous clients and servers can suddenly become interoperable. Microsoft .NET clients and Java clients can invoke EJBs exposed through SOAP.

WSDL is an XML technology that describes the interface of a web service in a standardized way. WSDL standardizes how a web service represents the input and output parameters of an external invocation. WSDL allows disparate clients to automatically understand how to interact with a web service.

UDDI provides a worldwide registry of web services for advertisement, discovery, and integration purposes. Business entities use UDDI to discover available web services by searching for names, identifiers, categories, or the specifications implemented by the web service. UDDI provides a structure for representing businesses, business relationships, web services, specification metadata, and web service access points.

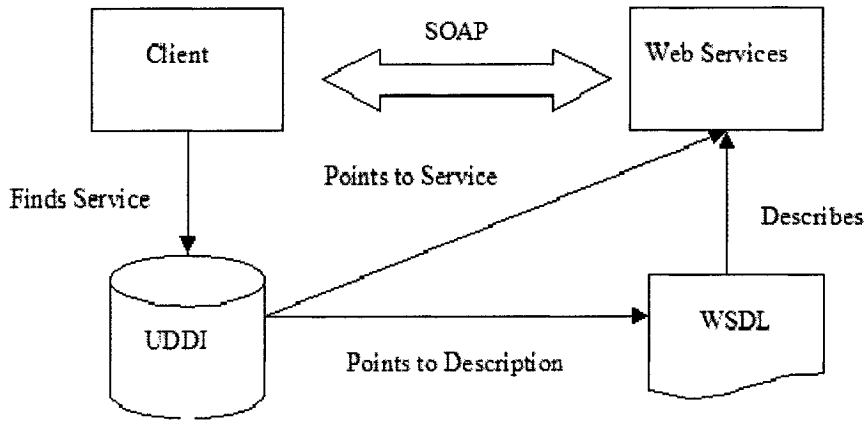

Figure 3.6 Web Services Interaction

Figure 3.6 shows when a service consumer (Web Services client) needs to locate another application (web services server) located somewhere on the network. The consumer queries a UDDI registry for the service either by name, category, identifier, or specification supported. Once located, the consumer obtains information about the location of a WSDL document from the UDDI registry. The WSDL document contains information about how to contact the web service and the format of request messages in XML schema. The consumer creates a SOAP message in accordance with the XML schema found in the WSDL and sends a request to the server application.

To sum up, in the first generation of middleware infrastructure, client objects can access service objects that are distributed on remote servers via IIOP or RMI protocols. CORBA can be considered the hyper-middleware solution, while both Sun and Microsoft have their own understandings and implementations of this object middleware architecture, which greatly improved the efficiency of development and deployment of enterprise applications by providing sets of system level services such as security, transaction and scalability. One important difference is that Microsoft .NET is a product strategy, whereas J2EE is a standard to which products are written. By using ubiquitous web protocols and universally accepted data formats, Web Services streamline enterprise

application communications, which was once a crucial problem caused by component technologies using proprietary protocols.

3.2 E-commerce Testing Challenges

Developed applications are not final products. They need rigorous testing before release and go live. It is especially true for e-commerce system as they held unique characters and risks that lead to catastrophic consequences. We explore e-commerce testing challenges from a business and a technology point of view.

3.2.1 Business Issues

A successful e-commerce application aims at the following goals:

- Usable. This is perhaps the most obvious but often neglected issue. Customers will simply leave the site if they cannot quickly and easily access or navigate a web page. This is a direct loss of customers. The root of the problem could be: incompatible browsers and platforms, poor web page functionality or navigation, dead hyperlinks or plug-in dependency.
- Secure. Security is a major barrier to e-commerce. With the rise in credit card fraud and ever-rising hacker attacks, customers increasingly avoid e-commerce sites and systems they perceive as insecure [Udo01]. Privacy, access control, authentication, integrity and non-repudiation are big issues in security regard.
- Scalable. The growth of electronic commerce has resulted in high demands on web sites that provide such services. As a consequence, servers often get overloaded and the quality of the services they provide decreases. Furthermore, overloaded communication channels are responsible for a significant portion of the overall user waiting time, regardless of the efficiency of the servers.
- Flexible. E-commerce systems will inevitably change due to new customer needs and business modifications. E-commerce applications must be able to offer flexibility by providing functional extensions and customizations or creating third party modules to provide additional features.

- Highly reliable and available. Businesses expect 24×7 availability to their customers. Failure or downtime is too expensive for critical e-commerce systems to tolerate. Unavailability equals lost revenue, business's reputation and customers. Several factors can influence availability, such as hardware reliability, software reliability, the effectiveness of load balancing, and the database's ability to handle concurrent users. Before going live, predicted business usage patterns should indicate maximum stress levels. Also, a backup and recover plan is necessary to prepare for the unexpected.

E-commerce applications integrate high value, high risk, high performance business critical systems, and it is these characteristics that enforce e-commerce applications must be undergo thorough and rigorous tests in order to determine the success of e-commerce at the business level.

3.2.2 Technical Issues

We can classify an e-commerce system into three distinct parts to discuss the relevant testing issues that are involved in each part: a front end system represents the human-computer interface, a back end system served as data and resources for the system and the middleware system contains the business logic and integrating software to link all the relevant software applications.

Front End Systems

Front end system testing could be further classified into static and dynamic testing.

Static Testing: The front end of an e-commerce site is usually a web site that needs testing in its own right. The site must be syntactically correct, which is a fairly straightforward issue, but it must also offer an acceptable level of service on one or more platforms, and have portability between chosen platforms. It should be tested against a variety of browsers, to ensure that images seen across browsers are of the same quality. Usability is a key issue and testing must adopt a user perspective. For example, the functionality of buttons on a screen may be acceptable in isolation, but can a user

navigate around the site easily and does information printed from the site look good on the page? Many of these tests can be automated by creating and running a typical user interactions test file, it will be useful for regression testing and to save time in checking basic functionality.

Dynamic Testing: Applications attached to an e-commerce site, either by CGI programming or server extensions, will need to be tested by creating scenarios that generate calls to these attached applications, for example by requiring database searches. The services offered to customers must be systematically explored, including the response time for each service and the overall server performance. This, too, must be exercised across alternative platforms, browsers and network connections. E-commerce applications are essentially transaction oriented, based on key business processes, and will require effective interactions between intranet-based and extranet-based applications.

Back End Systems

The back end of e-commerce systems will typically include ERP (Enterprise Resource Plan) and database applications. Back end testing, therefore, does not pose any problems from the business logic perspective, but there are potential technical problems, such as server performance and load balancing. The back end may well prove to be a bottleneck for user services, performance under load and scalability are key issues to be addressed.

Performance testing should monitor the deployed application in real time and alert operations groups to solve problems before users experience them. One way of carry out the work is by recording business processes that are executed in the applications, and constantly replaying these business processes against the applications, on a pre-defined schedule. Upon replaying these business processes, application performance is measured from the true end-user experience.

Load testing is used to predict system behavior and performance under the stress of a heavy user load. It exercises the entire enterprise infrastructure by emulating thousands of users and employs real-time performance monitors to identify and isolate problems.

Once the initial load test completes, tune the system if possible. During the tuning process, isolated system tests can be performed. This allows stressing a specific system to optimize its performance. If a system requires tuning, rerun the end-to-end procedure to determine the overall system capacity and to ensure the tuning produced the expected results.

Middleware and Integration

The process of developing an e-commerce site is significantly different from developing a web site. To facilitate commerce functions such as transaction, scalability and security an e-commerce system adds extra levels of complexity. A typical e-commerce application is being built that uses a database server, web server and payment server from different vendors, there is considerable effort involved in networking these components, understanding connectivity-related issues and integrating them into a single development environment. If legacy code is involved, this will add a new dimension to the problem, since time will need to be invested in understanding the interfaces to the legacy code, and the likely impact of any changes.

Integration testing is the key to e-commerce systems at the middleware level. Correctly functioning at the system back-end and front-end offers no guarantees of overall reliable functionality or performance. End-to-end testing of complete integrated architectures, using realistic transactions, is an essential requirement.

3.3 J2EE Development Process and Testing Concerns

3.3.1 J2EE Development Process

Enterprise applications are fast moving to Java 2 Enterprise Edition as it offers scalability, end-use flexibility, shorter time to market, and the comfort of working with open standards. But writing J2EE applications involves iterations consisting of development, assembly, deployment and running. The development cycle is based on "Roles" which envisions developers specializing in tasks of creation, assembly and

deployment. Before we introduce some of these roles that are relevant to the implementation phase of a J2EE system, it is necessary to take a brief look at the J2EE components (See Figure 3.7) as a reference to understand the objects different roles working on.

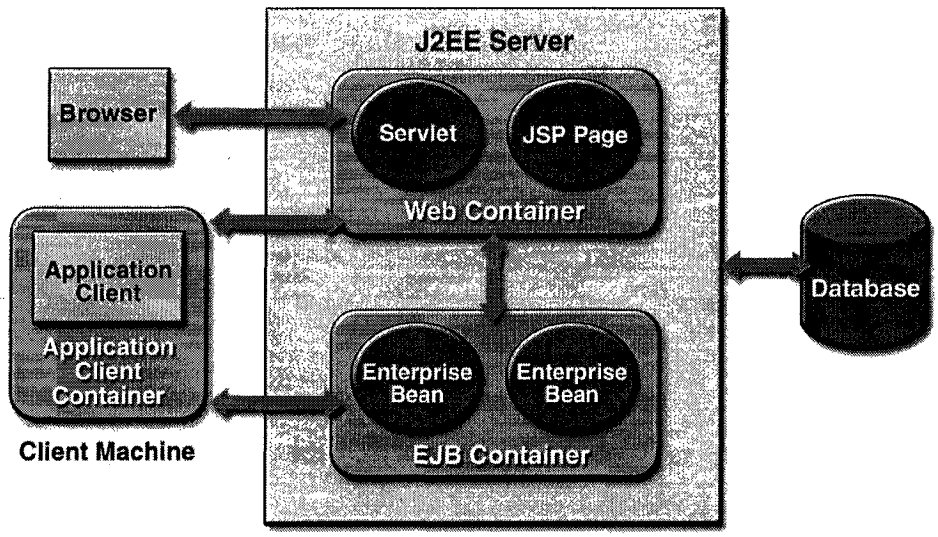

Figure 3.7 J2EE Components

Application Component Provider Role

The application component provider creates Web components, enterprise beans, applets, or application client for use in J2EE applications. Application component provider could be categorized into 3 sub-categories according to different component types:

- Enterprise Bean Developer: An enterprise bean developer writes and compiles the EJB source code, specifies the deployment descriptor and bundles class files and deployment descriptor into an EJB JAR file.

- Web Component Developer: The tasks for a Web component developer is very similar to an Enterprise Bean Developer: writes and compiles Servlet source code, writes JSP and HTML files, specifies the deployment descriptor for the Web

component, bundles class files, JSP files, HTML files, and deployment descriptor files in the WAR file.

- J2EE Application Client Developer: Again, this role is similar to a web component developer: Writes and compiles the client application source code, specifies the deployment descriptor for the client, bundles binary class files and deployment descriptor in the client application JAR file.

- The Application Assembler Role: The Application Assembler is the person who takes a set of J2EE modules and assembles them into a complete J2EE application, packaged as an Enterprise Archive. They are also responsible for providing instructions describing external dependencies of the application, making it possible for the deployer to install the application into the production environment.

- The Deployer Role: The deployer will take the packaged application and install it into the production environment. They are usually experts in a specific production environment. The deployer's job consists of three tasks: enterprise archive files installation, Security, user group, JNDI and application client manifest configuration, and at last, starting up the newly installed and configured application.

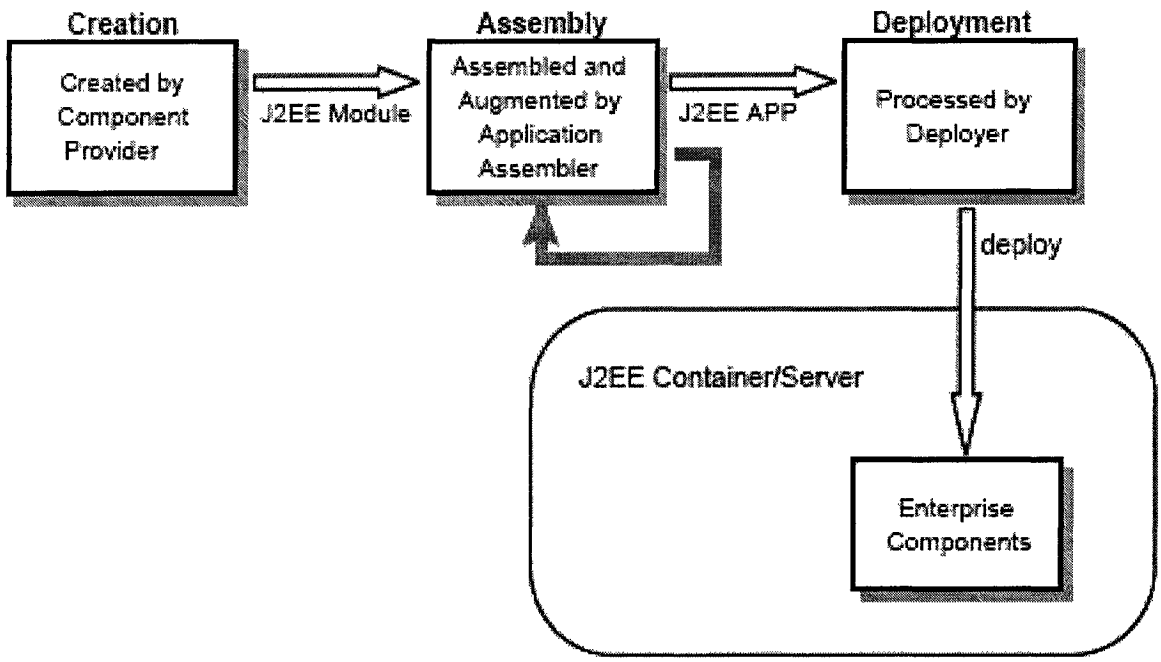

Figure 3.8 J2EE Development Lifecycle

Figure 3.8 depicts the development process lifecycle [J2EE_spe] including creation, assembly and deployment that involve the cooperation of different roles we explained above. Assembly and Deployment are key iterative tasks in all J2EE application development cycles. Usually, a single developer plays these roles in order to achieve faster cycles. And once the components are packaged, deployed and tested, they are integrated with the larger project assembly and final deployment. Usually, assembly and deployment preparation are daunting tasks because they involve writing the crucial XML-based deployment descriptors. These are the blueprint for a J2EE application and contain key information to collaborating beans and resources. Assembling involves grouping various components of an application into Java archives along with individual XML descriptors that describe the group. The descriptors grow non-linearly large with the archive. Deployment is also application server vendor-specific. The deployer must provide all information needed by the server to successfully run the application. As we can see J2EE development process is usually complex even for medium-sized applications. In chapter 5 we will introduce a cost-effective way to simplify this complex and error-prone process by using open source tools.

3.3.2 J2EE Functional Testing Concerns

Due to the inherent complexity of software developed with Java, particularly systems based on the J2EE platform, testing often proves more difficult. J2EE application development involves a wide range of technologies including Java Server Pages, Servlets, Enterprise JavaBeans and relational databases, therefore, J2EE application testing need to apply a wide range of testing techniques and tools.

J2EE Web application testers face unique testing challenges in several areas: first, J2EE software is typically distributed across several types of logical components such as Web servers, EJB application servers and database servers. These logical components are then distributed onto physical processors that are often organized into distinct of machines for scalability and performance purposes. To test distributed software, testers need tools and techniques that enable them to run both single-machine and cross-machine tests. For example, a typical end-to-end test may originate in a browser and connect to a Web server to access a Servlet, which interacts with session beans that may access the database through Java Database Connectivity (JDBC), and/or interacts with entity beans that access the database through the EJB persistence container. The beans produce a result that the Servlet passes to a JSP to produce HTML that can be displayed in the browser. To make matters worse, many aspects of J2EE are encapsulated, for example, the internal workings of EJB persistence container; therefore, testers are often limited to black box testing techniques.

In addition to an effective testing tool suite, testers also need a software environment that reflects the realities of testing under the compressed time and environmental constraints of Web application development. For example, in many organizations, it's quite common for development environments to differ from production environments: Perhaps developers are working on Intel-based machines running Windows NT, whereas the production environment is made up of Sun servers running Solaris. One solution for this problem could be setting up a staging area that serves as a testing ground. This enables

development teams to determine how a system is likely to work within the production environment without putting actual production systems at risk.

In general, software testing is hard. Testing object-oriented software often proves more difficult than structured and procedural software because object technology is used to address more complex problem spaces. Distributed technology adds a new dimension to the problem and is harder to test than non-distributed technology because of the additional complexities inherent in multi-node deployment. J2EE is a distributed, object-oriented software development platform, which means that it's one of the most difficult platforms to test.

3.4 Web-Based Application Functional Testing Best Practice Strategies

Figure 3.9 is an overview of our approach. As part of our cost-effective testing approach, Web-based application functional testing best practice strategies are adopted in the test case design. The general purpose of software functional testing is to verify if the product performs as expected and documented. Developers start creating a new product from a functional specification, which describes the product's capabilities and limitations. Testers utilize this specification as a guideline for expected product responses. Testing tasks are performed to validate specific features or functions, and the results of the tasks are verified against the expected response.

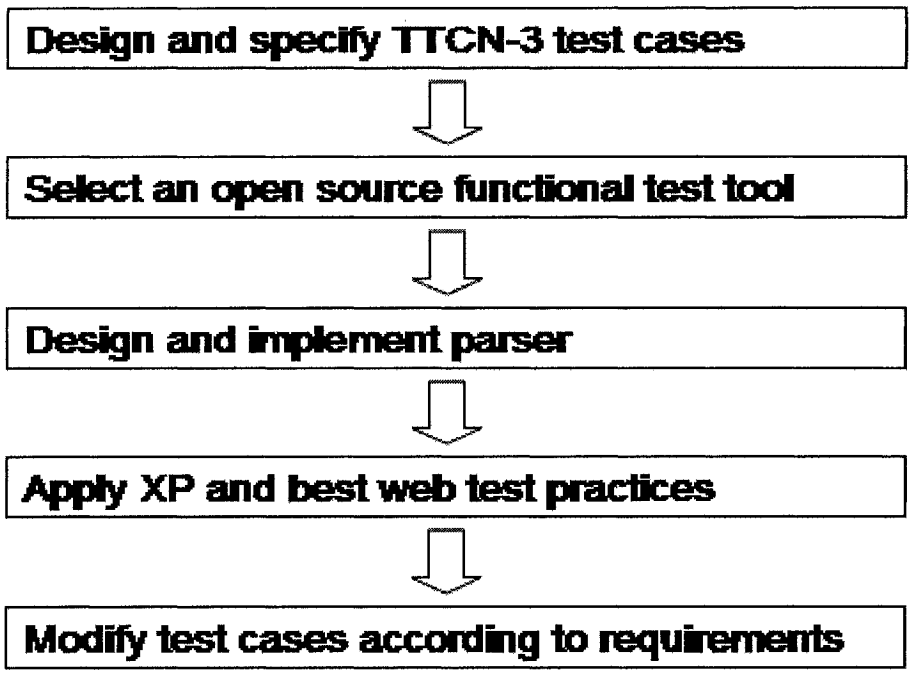

Figure 3.9 Web-based application best practices comprise step 4 of our approach

Functional test includes a broad category of test methods. Since we are interested in using best test practices, we adopt the following test strategies in [Nguyen03].

FAST (Functional Acceptance Simple Test)

FAST testing focus on the lowest level of functionality of a software program. It is a broad yet shallow black-box type of testing. In a web based system, FAST may include the following tests: Links, basic controls such as navigating and refreshing, and simple action command checks such as add, remove, update submissions. FAST testing simply verifies that the item is handled, not whether the function is correctly provided.

FET (Forced Error Test)

FAST tests try to prove that certain functionality is available under normal conditions. FET tests however, deliberately put the system into an error condition and try to find if the system can recover gracefully or whether the application exits without data corruption and with an opportunity to preserve work in progress.

TOFT (Task Oriented Functional Test)

A task may compose of a series of individual operations in a timely fashion. TOFT is used to check that a certain task is achieved by integrate and perform necessary individual operations in order. Tasks often represent a list of system features or functions that are specified in the system specifications and requirements documents. A features-to-be-tested list is extracted from these documents and a TOFT test case is defined according to an entry in that list.

Boundary Test

As the name suggests, Boundary tests focus on the boundaries of each variable. An example may be in a login form, user name length may be limited to 1 to 6 characters. A Boundary test could select 0, 1, 2, 5, 6, 7 characters as user names to try to login and expect the system to successfully login with 1, 2, 5, 6 character user names, and pop up error messages when attempts are made to login using 0 and 7 character user names.

FAST tests are used as an entry criterion for Functional Testing. FET tests and Boundary tests are high-yield (high expectation of defects) and check robustness of the user interface as well as proper error handling. Finally, TOFT tests are extremely important, customer-oriented test scenarios with a specific customer goal in mind.

Chapter 4 Formal Method Testing with TTCN-3

This chapter presents the international standard test language-TTCN, its history and application areas. Our focus will be on TTCN-3, which is a totally re-designed test language compare to its ancestors. The first section introduces formal methods and Formal Description Techniques (FDTs) as the origin of TTCN while the following sections give descriptions and justifications of using TTCN-3.

4.1 Formal Method and FDT

With the increasing complexity of today's software systems and their greater impact on our daily life, software testing becomes one of the most critical and strategic processes in the software development lifecycle to provide reliable systems. In general, software testing has two major drawbacks. First, it is time consuming in terms of writing and executing test cases. Secondly it is error prone either by incorrectly specified test cases and missing test cases for key functional or non-functional requirements. How to overcome these problems of software testing?

"One way of achieving this goal is by using formal methods, which are mathematically-based languages, techniques, and tools for specifying and verifying such systems. Use of formal methods does not priori guarantee correctness. However, they can greatly increase our understanding of a system by realign inconsistencies, ambiguities, and incompleteness that might otherwise go under detected. "[Clarke96]

The motivation for using formal specification for testing is to reduce the overall time spent on testing and make the testing process more rigorous. In a typical software development cycle, testing takes a significant proportion of the time [Boehm81]. The benefits behind formal methods is that time spend on specification and design will be repaid by a higher quality of product, that is aims to increase the understanding of a

system by revealing errors or aspects of incompleteness that might be very expensive to rectify at a later date. Other benefits of applying formal methods techniques include: aiding understandability of requirements, clarifying specifications and helping to ensure consistency with other standards.

4.2 FDT languages and tools

To support the FDT approach, a series of standardized FDT tools and languages are developed by ETSI (European Telecommunication Standard Institute) and other international institutions, such as International Organization for standardization (ISO) and the International Consultative Committee for Telephony and Telegraphy (CCITT now ITU-T). FDT tools and languages applied to various software development processes. We present UML and SDL as examples in this field.

4.2.1 UML

The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.

Each UML diagram is designed to let developers and customers view a software system from a different perspective and in varying degrees of abstraction. UML diagrams commonly created in visual modeling tools like Use Case Diagram (see Figure 4.1) that displays the relationship among actors and use cases, and Class Diagram that models class structure and contents using design elements such as classes, packages and objects. Interaction Diagrams/Sequence Diagram is used to display the time sequence of the objects participating in the interaction. These diagrams consist of the vertical time dimension and horizontal object dimension. Collaboration Diagram displays an interaction organized around the objects and their links to one another.

The UML is an important part of object oriented software development process. It uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

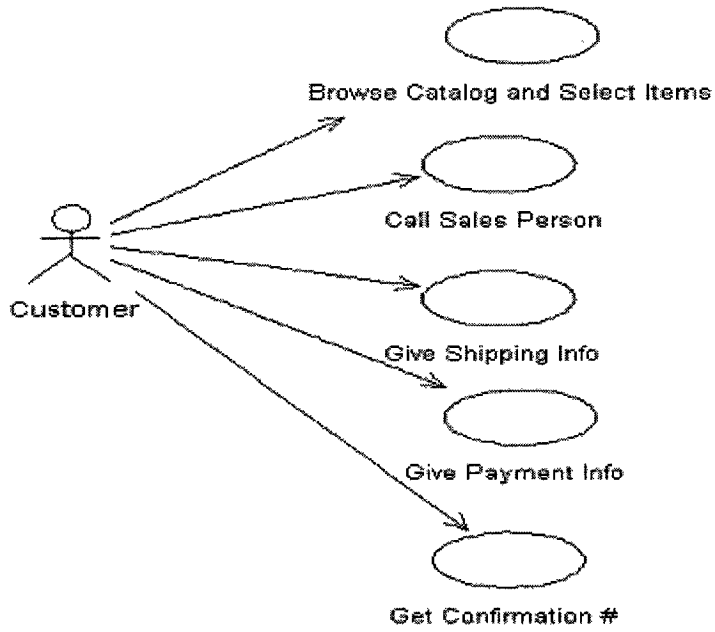

Figure 4.1 Sample Use Case Diagram

4.2.2 SDL

Specification and Description Language (SDL) was issued in 1976 in CCITT for the telecommunications industry. The latest versions expanded the language is standardized by ITU in its Z.100 series.

The basic theoretical model of an SDL system consists of a set of extended Finite State Machines (FSM) that run in parallel. These machines are independent of each other and communicate with discrete signals. SDL describes a system by means of blocks that are connected by channels. Each block may contain substructures of blocks or process sets that are connected by signal routs. Processes execute concurrently with other processes and communicate by exchanging signals or remote procedure calls. The request and

reception of signals are the events that trigger state transition in the behavior processes. Figure 4.2 shows a sample SDL system, block and process view.

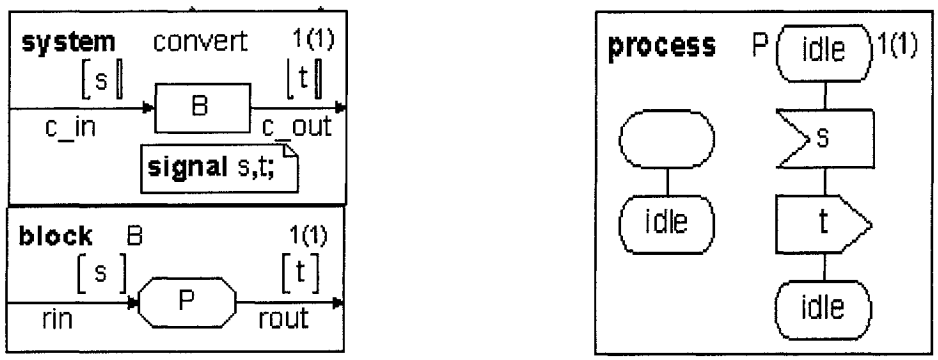

Figure 4.2 Sample SDL System, Block and Process View

SDL is capable of being the core of full-scale projects because of its abilities to interface with other languages, such as UML object models for analysis, as well as abstract system notation one (ASN.1) or CORBA/ IDL (Interface Description Language) data-type definitions. Also, there are tools available to generate C/C++ executable code directly from the SDL design. Tests can also be generated from the SDL specification by making a test suite in TTCN-2. See Figure 4.3 for the relations between these languages. [Telelogic]

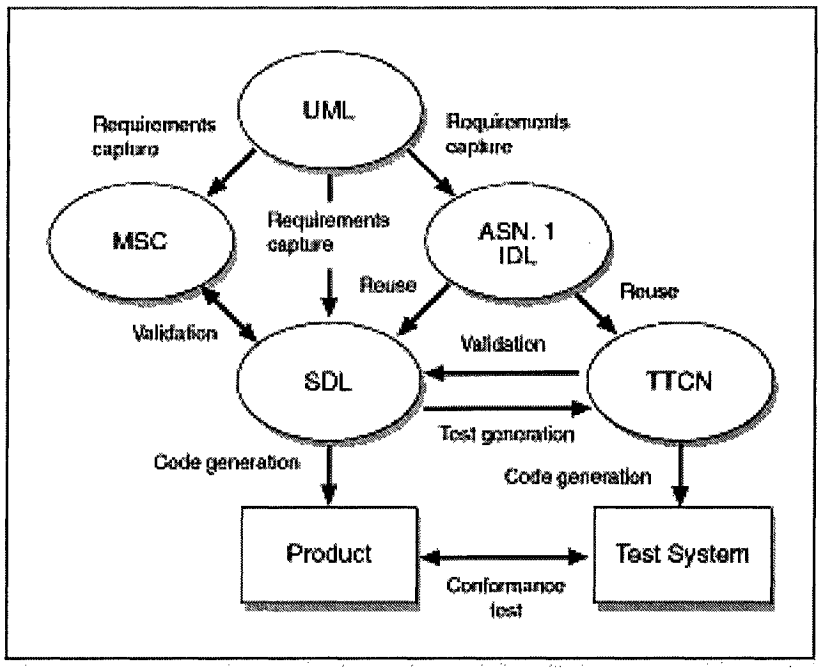

Figure 4.3 Relations between Different Languages and SDL

4.3 TTCN

The early version of TTCN is the acronym for Tree and Tabular Combined Notation and it is originally defined and standardized in Part 3 of the international standard 9646 OSI Conformance Testing Methodology and Framework (CTMF) [Con98] which uses TTCN test cases to describe sequences of stimuli of an implementation under test (IUT) to verify the expected responses are conformance with the specification.

In its latest generation- TTCN-3, standardized by ETSI (ES 201 873 series) and the ITU-T (Z.140 series), the language has been expanded to become a true, multi-purpose test language with a new acronym as Testing and Test Control Notation. TTCN-3 is used to write detailed test specifications in many kinds of test applications, including mobile communications, wireless LANs, cordless phones, broadband technologies (B-ISDN, ATM), CORBA-based platforms and Internet protocols such as IPv6, SIP and OSP. We will explore a TTCN-3 application for Web Service testing in the up coming section.

4.3.1 Introduction of TTCN-3

TTCN-3 has the look and feel of a modern programming language. This general purpose, flexible and user-friendly test language is easier to learn, easier to use, and easier to implement. We will describe some the main characters of this language bellow.

Presentation Formats

TTCN-3 offers three presentation formats (See Figure 4.4). The textual core format which acts not only as a generic text based test language but the interchange format between TTCN tools and provide semantic basis for other presentation formats, the tabular format that is developed from TTCN-2 in conformance testing and the graphic format which is derived from MSC is named Test Sequence Chart (TSC). The graphic format is to visualize the communication between and within test systems and it gives a clear and intuitive understanding of test behavior.

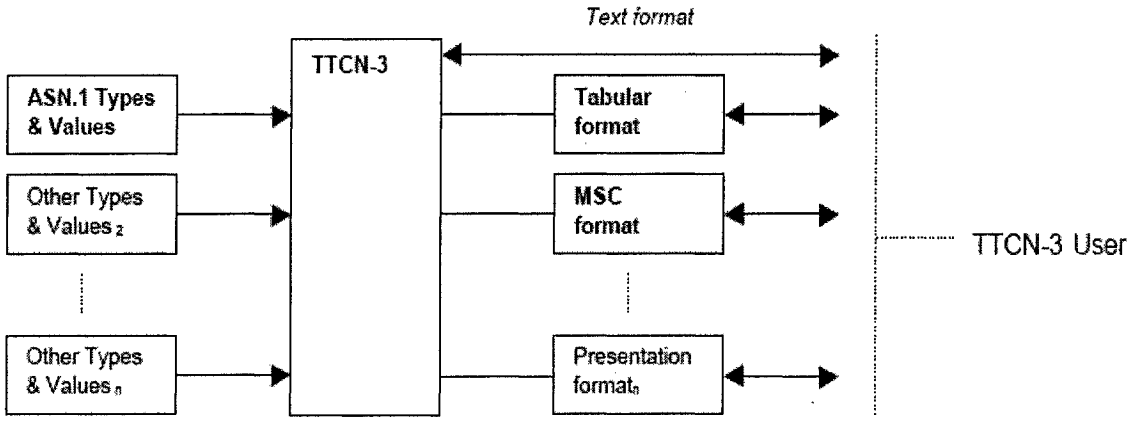

Figure 4.4 TTCN-3 Presentation Formats

Modules

The top-level unit of TTCN-3 is the module. A module may import definitions from other modules. A TTCN-3 module consists of a definitions part and a control part. The module definitions part specifies test components, communication ports, data types, constants, test data templates, functions, remote procedure call signatures, named alternatives and test cases. It is possible to re-use top-level definitions specified in other TTCN modules by using import statements. TTCN-3 has no explicit export construct and thus, by default,

all definitions in the module definitions part may be imported by other modules. As shown in Figure 4.5, it is possible to import single definitions, all definitions of a module, groups of definitions and definitions of the same kind.

```
import type MyType from MyModuleA; // imports a single definition
import all from MyModuleB; // imports all definitions of a module
import group MyGroup from MyModuleC; // imports a group
import all type from MyModuleD; // imports all type definitions
```

Figure 4.5 Examples of Import Construct

The control part of a TTCN-3 module is not mandatory. It is just the starting point of executing test cases specified in the module definitions part. Variables, constants or timers may be declared and program statements may be used to specify the selection and or repetitious execution order of individual test cases. Variables defined in the module control part are local, i.e., they cannot be accessed by functions or test cases called inside the control part. TTCN-3 does not support global variables. If required, variable values can be passed into test cases and functions as parameters.

Test Configurations

TTCN-3 allows the specification of concurrent test configurations. A test configuration consists of a set of interconnected test components with well-defined communication ports and an explicit test system interface that defines the borders of the test system. Within every test configuration, there is one Main Test Component (MTC). All other test components are called Parallel Test Components (PTCs). The MTC is created automatically at the start of each test case execution and the behavior defined in the body of the test case is executed on this component. During execution of a test case, PTCs can be created and stopped dynamically by the explicit use of create and stop operations. The conceptual view of a typical TTCN-3 testing configuration is shown in Figure 4.6.

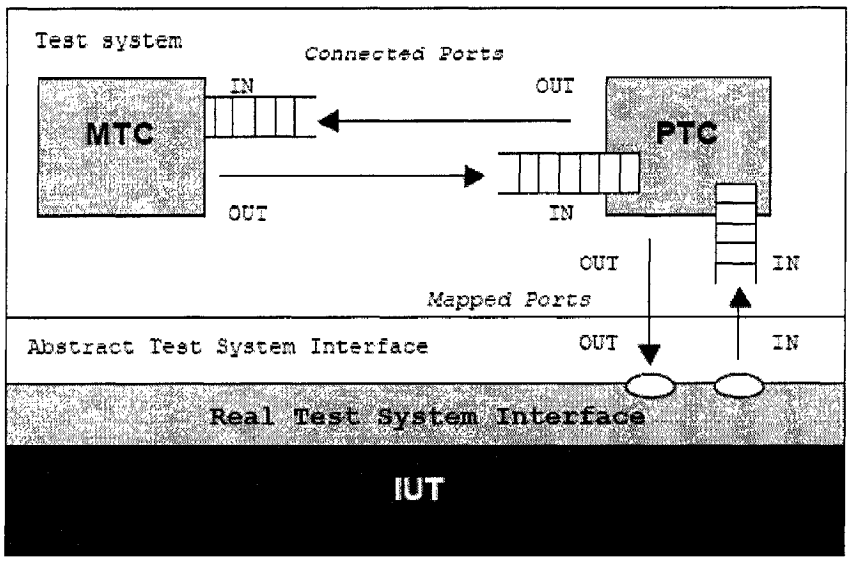

Figure 4.6 TTCN-3 Test Configurations

Communication Port Types

Ports facilitate communication between test components as well as between test components and the test system interface. Each port is modeled as an infinite FIFO queue that stores the incoming messages or procedure calls until they are processed by the component owning that port. TTCN-3 ports are either message-based or procedure-based. Message-based ports are used for asynchronous communication by means of message exchange. Procedure-based ports are used for synchronous communication by means of remote procedure calls. Ports are directional and each port may be defined as "in", "out" or "inout" which stands for in direction, out direction or both directions of messages or procedures respectively. Figure 4.7 is an example port definition.

```
type port MyMessagePortType message {
in MsgType1, MsgType2;
out MsgType3;
inout integer
}
```

Figure 4.7 TTCN-3 Port Type

Test Cases and Functions

The test cases are the probes that have to be executed in order to judge whether an implementation under test passes the test or not. Test cases are defined in the module definitions part and called in the module control part. Each test case returns a test verdict of either none, pass, fail, inconclusive or error. This means a single test case can be considered to be a special kind of function returning a test verdict. The test case defines the behavior of the MTC and will be started automatically when the test case is called. The MTC type is required to make the port names of the MTC visible inside the behavior definition. The type of the system interface is mandatory, if during the test run several test components are created and stopped dynamically. If the MTC performs the whole test on its own, the type of the test system interface is identical to the MTC type and can be omitted.

In TTCN-3, functions are used to express test behavior or to structure computation in a module, for example, to calculate a single value or to initialize a set of variables. If a function defines test behavior, the type of the test component on which the behavior is executed has to be specified by means of a "runs on" clause. This type reference makes the port names of the component type visible inside the behavior definition of the function. This is shown in the definition of function MyBehaviour in Figure 4.8.

```
function MyBehaviour (inout integer MyPar)
runs on MyPTCType
{
var integer MyVar := 5 * MyPar;
PCO1.send (MyVar);
...
}
```

Figure 4.8 Example of Function Definition

Communication Operations

TTCN-3 supports Message-based (asynchronous) and Procedure-based (synchronous) communication. As illustrated in Figure 4.9 asynchronous communication is non-blocking on the send operation, where processing in the MTC continues immediately after the send operation. The IUT is blocked on the receive operation until it receives the send message.

Synchronous communication in TTCN-3 is related to remote procedure calls. As sketched in Figure 4.10, the synchronous communication mechanism is blocking on the call operation, where the call operation blocks processing in the MTC until either a reply or an exception is received from the IUT. Similar to the asynchronous receive operation, "getcall" blocks the IUT until the call is received.

Figure 4.9 Asynchronous Send and Receive Operations

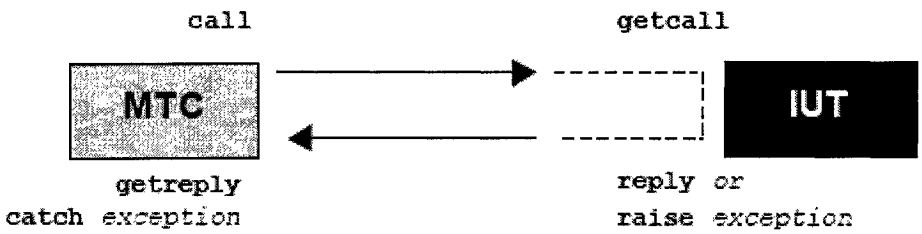

Figure 4.10 Complete Synchronous Calls

It is impossible to give a detailed and complete description of TTCN-3 in this thesis. We show some interesting features to distinguish this test language as the standard testing approach. Further information can be found in [ETSI01].

4.3.2 TTCN-3 Application Example -TTCN-3 for Web Services Testing

We mentioned at the beginning of this section that TTCN-3 is not restricted to conformance testing. It can be used in many areas, for example: interoperability test, robustness test, performance test, regression test, system test and integration test. We use TTCN-3 for web service test [Schieferdecker03] as an example to show its application in the real world.

Web Services is the development architecture for middleware applications. Web Services testing include SOAP communication protocol, WSDL data format and UDDI discovery service. The test procedure is depicted in Figure 4.11.

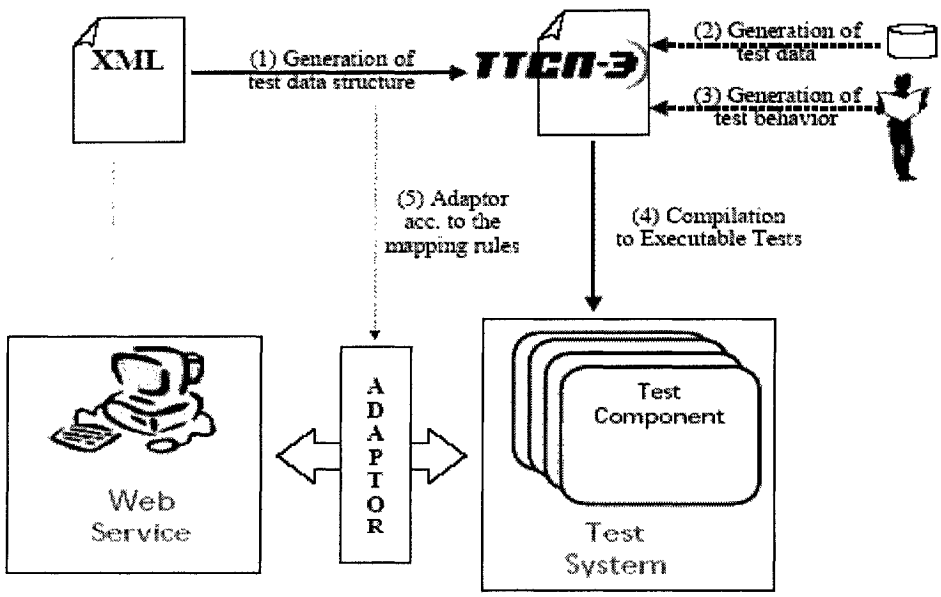

Figure 4.11 TTCN-3 Web Service Testing Process

As discussed earlier (Communication port types), TTCN-3 is designed to support ports as a fundamental type. Therefore, it is straightforward to map stimuli and responses in a Web-based system to ports using TTCN-3. Concurrency is easily handled in TTCN-3 by using user-defined test components to perform test procedures in parallel. Also, TTCN-3 is a modular language and has a similar look and feel as a typical programming luggage. A TTCN-3 test specification uses type definitions for test data structures, templates to define concrete test data, function and test case definitions for test behavior and control definition for execution of test cases. Test data is derived from an XML definition, and test behavior is specified according to the request/response sequence of the service. Finally, the test module is compiled into executable code. To facilitate aforementioned test process, the authors implemented the following tasks: mapping XML DTD and XML schema to TTCN-3 for generating test data structures and generating test configurations using message-based communication in TTCN-3 to specify the request and response behavior. In term of performance and load testing, the authors set up various sets of PTCs (Parallel Test Components) to simulate the different interfaces to the web service requests and a MTC (Master Test Component) to control the dynamic creation of these test components. Actual test data is represented in the TTCN-3 template structure and some basic system services are defined in TTCN-3 function structure.

The authors conclude that testing web service using TTCN-3 improved the development efficiency by preventing late detection of errors that typically requires complex and costly repairs. Furthermore, the automated test approach helps in particular to efficiently repeat tests whenever needed for new systems.

Chapter 5 Our XP approach to Test Java Applications and E-commerce Systems

This is the critical part of this thesis work. Extreme programming is the principal directing our test practice. We introduce the advantage of this simple yet powerful testing methodology including open tools that are widely used in the Java world. The Big Picture section presents our approach of applying XP to test e-commerce systems based on J2EE platform including the generation of TTCN-3 test scripts. Finally, we explain the design and justification of the parser that translates TTCN-3 test script to Java test code.

5.1 Extreme Programming

We presented extreme programming in general as an emerging popular software development methodology in Chapter 2. In this section, we briefly review this software development methodology.

According to the pioneer of extreme programming Kent Beck's definition [Beck99]: "Extreme Programming (or XP) is a set of values, principles and practices for rapidly developing high-quality software that provides the highest value for the customer in the fastest way possible. XP is extreme in the sense that it takes 12 well-known software development "best practices" to their logical extremes."

Extreme programming is not only simply about programming techniques but is a complete approach to software development, including strategies for planning, gathering user requirements and everything else necessary to develop complete applications. These strategies are summarized briefly below in its four values and 12 common practices.

5.1.1 Values

In order to understand XP practices, we must first look at the values on which they are based. Everything in practices must be interpreted in the light of these four values.

Simplicity

Simplicity means treating every problem as a simple problem [Jeffries00]. This principal applies to all stages of software development process. Focusing on simple designs minimizes the risk of spending a long time designing complicated frameworks that the customer may not want. Keeping code simple makes changing code easier as requirements will change inevitably. Most importantly, focusing on simple solutions to today's problems minimizes the cost of change over time.

Communication

Communication in extreme programming comes in many forms. Source code comments and self-documenting code facilitate the communication between code and programmers. By setting up a concrete example of how to exercise a class's functionality, unit tests also act as a critical part of the system's documentations. Communication between programmers comes in the form of "pair programming". The customer and programmer also communicate constantly by means of an on-site customer conveying the requirements to the team. The customer decides which features are most important, and is always ready to answer questions.

Feedback

Feedback is facilitated in the form of on-site customer, small releases and unit tests.

As stated in the above communication section, on-site customer offers an immediate feedback about the project. With a short release cycle, the customer is able to evaluate each new feature as it is developed and minimizing the necessity to rework and helping the programmers focus on what is most important to the customer. Code can offer feedback to programmers. In XP, unit tests give immediate feedback on the quality of the code. After each change to the source code, programmers run all of the unit tests. A broken test provides immediate feedback (that is the most recent change caused

something in the system to break). After fixing the problem and verifying the fix, programmers check the change into version control systems and rebuild the entire system.

Courage

Based on the above values and the core practices, XP programmers have the courage to improve the project by refactoring the old system and throwing away any useless code. The unit test suite they built along with the development process works as the backbone to support programmers monitoring any bug introduced by the change to the system. The confidence to change an old system encourages programmers to move quickly and at the same time leads to better project quality.

5.1.2 Core Practices

In order to facilitate the 4 values, the extreme programming methodology utilizes 12 core practices, namely, on-site customer, metaphor, small releases, planning game, pair programming, collective ownership, testing, refactoring, simple design, continuous integration, coding standard and 40-hour week. Most of these practices are not new in software development process. As Martin Fowler [Fowler00] explained: "Many of these practices are old, tried and tested techniques, yet often forgotten by many, including most planned processes. As well as resurrecting these techniques, XP weaves them into a synergistic whole where each one is reinforced by the others". In this section, we present two of these 12 core practices (automated testing and continuous integration) that are relevant to our approach.

Automated Testing

Testing is an important concern throughout the software development lifecycle and is accorded particular importance in extreme programming methodology [Beck98]. In XP, there are two categories of tests: unit tests and acceptance tests.

To help ensure that a suitable suite of unit tests is built throughout development, a coding practice known as "test-first" programming is implemented [Beck99]. Before adding a

new piece of functionality, a unit test exercising the non-existent code is written. Then, enough production code is written to allow the test to compile, but not pass. The unit test is executed to ensure that it really does fail because no functionality is implemented yet. Then production code is written until the test passes. This programming style helps ensure a consistent test suite is developed alongside the production code.

Acceptance tests are distinguished from unit tests in two ways. First, they should test the system end-to-end. Second, the customer is involved in creating the acceptance tests. Ideally, a framework should be built that allows the customer the full ability to add new tests for the system without any technical intervention.

Unit tests ensure each technical detail is working properly and acceptance tests ensure each customer requirement is working properly. Having a consistent test suite removes the fear of improperly revising previously written code, i.e., introducing bugs or breaking things at a late stage of a project. Programmers can make bold changes and see the result of the change immediately by re-running the test suite.

Continuous Integration

Integrating each programmer's latest code together can be a difficult process. Traditional approaches tend to postpone the integration until a lot of coding has been done. This Big-bang integration dumps a load of problems on the team all at once, and those problems often have hundreds of possible causes. XP promotes continuous integration. After writing new code that passes all unit tests, programmers must then integrate their changes with the latest code base and ensure all tests still pass. If there are failures, the cause for any particular integration is more obvious since the tests ran before, and so the new code must be to blame. Fixing is easier, takes less time, and keeps the team moving at maximum speed.

5.2 Use of Open Source Tools

Open Source software can be distributed freely and users are free to make changes to the source code and modify the program as they please. Many Open Source software

projects, such as Linux, Apache, and FreeBSD, are the work of a team of individuals from around the world. There are several benefits of open source tools:

(1) Free open source tools greatly reduced solution overhead;

(2) Open source tools are continually viewed, used, and modified by many users and developers [Infoworld03]. The quality of these tools will be improved along with their practices and inspections;

(3) Some of the open source tools are becoming the de facto tools in their application areas such as Apache Ant and JUnit. In this section we briefly review some of these open source tools that focus on Java applications.

5.2.1 Unit Test Tool

JUnit [JUnit] is the de facto standard for Java unit testing. It is a simple framework for creating automated unit tests. JUnit test cases are Java classes that contain one or more unit test methods, and these test are grouped into test suites.

JUnit tests are pass/fail tests explicitly designed to run without human intervention. Speed is important for JUnit tests, because as more tests are written and integrated into the build process, it takes longer to run the entire test suite. Programmers do not want to be interrupted for long periods of times while test run, so the longer the test take to execute the greater the likelihood programmers will skip this critical phase.

5.2.2 Integration Test Tool

Cactus [Cactus] is an open source test framework for server side Java code. It aims at testing Servlets, JSPs, and Servlet filters. Cactus tests execute on both client and server. The client side allows HTTP headers and HTTP parameters to be added to the outgoing request. The server side invokes Servlet methods, performs any necessary assertions and sends back a response to the client. The client may then check that the response contained the expected information. Figure 5.1[Cactus] shows the architecture of Cactus and how it works.

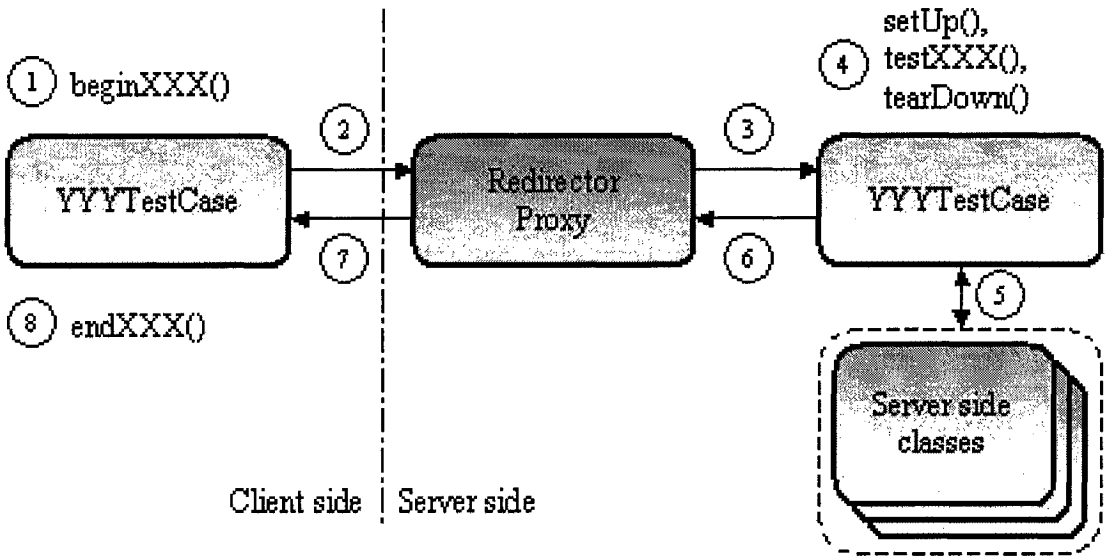

Figure 5.1 Cactus Architecture

5.2.3 Functional Test Tool

With unit testing, sometimes the big picture gets lost. Another important role in XP methodology is functional testing that enables developers to get feedback about the overall state of their system. Automated functional testing tools can save developers from the drudgery of manually inspecting areas of a site before the project goes live. HttpUnit [HttpUnit] is one of these tools.

The name HttpUnit may mislead people to think it is for unit test. In fact, HttpUnit is designed for functional or black box testing. Tests written using HttpUnit will query the Web server externally and analyze the responses received. HttpUnit in essence mimics a Web browser, and the HttpUnit API can emulate a number of browser behaviors, including Form submission, JavaScript, HTTP authentication and Cookies. You can also use the HttpUnit API to analyze the content returned when a Web page is loaded.

5.2.4 Performance Test Tool

Web projects need a proper tool to handle performance issues. Commercial performance test tools help pinpoint performance problems but typically require human intervention to run and interpret the results. These tools are not designed to execute automatically as part of a continuous integration process. JUnitPerf [JUnitPerf], on the other hand is a tool for continuous performance testing. The goal of such tests is to ensure that the code executes fast enough under different load conditions.

However, the common practice of JUnitPerf is combined with commercial tools. First, testers identify the bottleneck of the system by using commercial performance tools. Then a JUnitPerf test is setup to monitor the performance problem. Next, refactor the code that is causing the problem until the JUnitPerf test passes.

The performance test suite should be run automatically and independent of other JUnit tests. This will ensure that the overall execution of JUnit tests isn't affected by the additional time spent executing JUnitPerf tests.

5.2.5 Automatic Build Tool

From the description about J2EE project development process in Chapter 3 it is not hard to draw the conclusion that it is complex and error-prone. Building such a system manually is neither pleasant nor realistic. We need an efficient and repeatable way to tackle this problem. First, let us review the following tasks that could be part of the process. Basic tasks include:

- Compiling all source code
- Compiling test code
- Creating J2EE deployment units - WARs, EJB JARs, and EARs
- Generating Javadoc for the entire application
- Automatically detecting test cases and running tests in a single operation

More advanced tasks include:

- Checking code out of a version control system
- Tagging all files in the version control system with a specified build identifier
- Putting a database in the required state before a test run and cleaning the database after a test run
- Deploying onto an application server

All these tasks can be accomplished using Another Neat Tool (Ant) [Ant], now the de facto standard mechanism for Java build scripts. Ant is an open source tool from the Apache project designed to support software builds. An XML file called build.xml specifies which tasks Ant follows when building a project.

Ant is critical to any successful XP implementation. One can not expect a team of XP programmers to constantly refactor their code, run unit tests, and integrate their changes without a fast, predictable build environment. XP assumes that programmers write a lot of small, incremental pieces of code. Programmers must compile and run all unit tests after making each small change; therefore, the build needs to be fast. When builds are slow, programmers are discouraged from the constant refactoring and testing that XP demands. Ant facilitates its speed by not doing the work if a file is out of date, for example, code is only compiled when Java source files are newer than their corresponding class files. Also, ant tasks use a simple pattern-matching syntax to locate files quickly, allowing programmers to write build files that perform work on the correct subset of a source tree.

5.3 The Big Picture

Previous sections of this chapter described the general values and practices of extreme programming. In this section we set up a model (see Figure 5.2) that adopts some of the above XP methodology practices (automatic testing and continuous integration) and open source test tools (HttpUnit and Apache Ant) to exercise in a J2EE e-commerce application testing process. To facilitate formal method specification using TTCN-3, we implement a parser that translate TTCN-3 test scripts into Java test code which then could

be applied to functional test tools, in our case, HttpUnit. We first explain the J2EE development route and the testing route, followed by the design and implementation of the parser. A detailed case study that implements the tests is presented in Chapter 6.

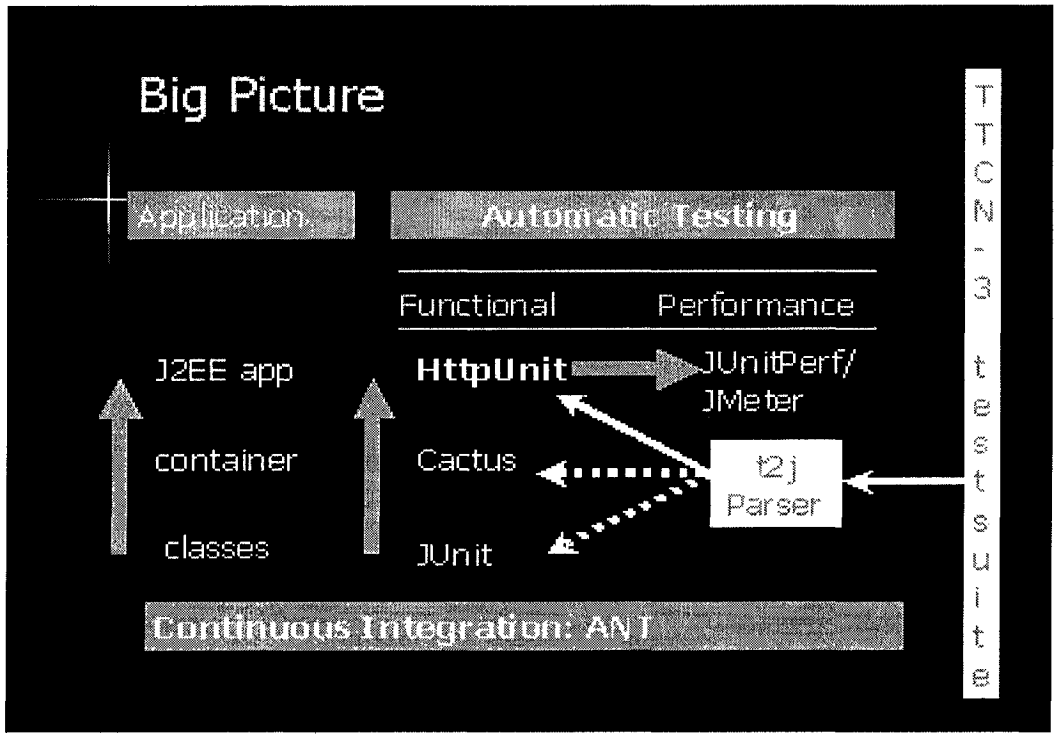

Figure 5.2 the Big Picture

5.3.1 Applying Automatic Testing and Continuous Integration

Two of the most valuable practices of XP are automated testing and continuous integration. We choose HttpUnit to achieve automatic web functional testing and use Apache Ant for continuous integration.

Automated Testing: HttpUnit

Designed and implemented by Russell Gold, HttpUnit [HttpUnit] is a free open source Java API (Application Programming Interface) for accessing web sites without a browser. It is ideally suited for automated functional testing of web sites when combined with a Java unit test framework such as JUnit. HttpUnit simulates the relevant portions of browser behavior, including form submission, basic HTTP authentication, cookies and

automatic page redirection, and allows Java test code to examine returned pages as text, an XML (Extensible Markup Language), DOM, or containers of forms, tables, and links.

Continuous Integration: Apache Ant

Continuous integration comes with a price of setup and configuration, but it is well worth the energy spent compared with the following benefits:

- Developers and customers could see the progress of the project so far
- Frequent and early integration reduce the integration pain
- Especially valuable for J2EE applications: the J2EE development model employs significant processes during assembling and deployment. For example, wrapping and packaging a full J2EE application requires in depth knowledge of different archive formats (JARs, WARs, and EARs), deployment descriptors, and methods of deploying applications and components into existing containers

We use Apache Ant's extended capability of integrating the test process into the overall integration process.

5.3.2 Development Process for J2EE Applications

The J2EE system development process consists of several stages including component creation and packaging, application assembly and application deployment.

First of all, the component provider creates components, like EJBs, JSPs and Servlets. When the components have been developed they are packaged into J2EE modules (client application Jar files, Enterprise Java Beans, web applications war files, etc.). Secondly, provided with complete J2EE modules, application assemblers combine these components into one J2EE application. They need to create the directory structure, edit module deployment descriptor, place the modules in their correct containers and package the application. Finally, application deployers deploy the application to the J2EE application server. Deployment typically involves using a platform's deployment tool to specify location-specific information, such as a list of local users that can access it and the name of the local database. Once deployed on the local platform, the application is ready to run.

5.3.3 Test Route Using Open Source Tools

The test route of a J2EE application goes along the development process and different test tools are applied to different stages of development. For example, unit test JUnit should be performed throughout the class building stage, every class should pass 100% unit test and a large number of unit test cases should be grouped to be a unit test suite working as both a trap net for bugs introduced by refactoring and a live example for implementation documentation. Cactus should be used at the container level to check that Servlets, JSPs and filters work within a container environment. Web functional tests using HttpUnit should be performed at regular intervals as part of the integration process to expose side effects introduced by refactoring and as a way of project progress measurement. Performance testing using JUnitPerf or JMeter can be performed after all functional tests passed.

5.3.4 Designing the TTCN-3 Test Script

When drafting TTCN-3 test scripts, we need to identify and represent customer interactions with the Web site; we use customer login use case as an example to explain TTCN-3 script creation process. The customer login use case could be presented as a use case scenario in either UML or as an MSC (see Figure 5.3).

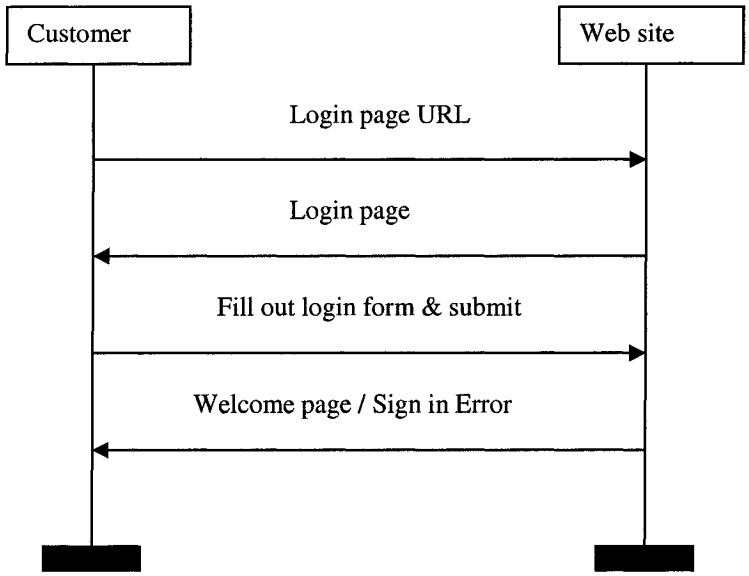

Figure 5.3 Message Sequence Chart for Customer Login Use Case

This will serve as the basis for TTCN-3 call signature definitions. For example, the "testLoginCall" signature contains the following information: URL of the initial page, login form name, login form field name and value and the submit button name. The "testLoginCall" signature could be defined as:

```
signature testLoginCall{myUrlType url, myFormNameType
fName, myFormType uName, myFormType pwd, mySubmitType
submit};
```

An example test call following the format of this signature is in the "testcorrectlogin" test case:

```
testcase testcorrectlogin runs on MTCType system
TestSystemType{
var PTCType loginptc;
loginptc.Create;
connect(loginptc:pco, system:tsipco);
```

```
map(loginptc:cp, mtc:cp);
pco.call(testLoginCall:{loginurl,loginformname,
correctusername, correctpassword, submit})
```

The returned page title is received in the "getReply" operation parameter, in the correct login use case. The returned page title should be a "Welcome" string as follows:

```
alt{
[]pco.getreply("Welcome"){
    verdict.set(pass);
    }
  []pco.catch{
    verdict.set(fail);
    }
stop;
}
```

Secondly, we identify other elements to be tested within Web pages such as URLs, forms and links. We define these elements in "type record" in TTCN-3, for example:

```
type record myUrlType{charstring url};
type record myFormNameType{charstring formname};
type record myFormType{charstring field};
type record myLinkType{charstring link};
type record mySubmitType{charstring submit}
```

Following the "type record" definition, we assign test values to these elements in "template" in TTCN-3:

```
template myUrlType loginurl:=
{"http://localhost:8000/petstore/signon_welcome.screen"};
```

```
template myFormNameType loginformname:=
{"existingcustomer"};

template myFormType correctusername:={"j_username", "bob"};

template myFormType correctpassword:=
{"j_password","bobpass"};

template mySubmitType submit:={"submit"};
```

Note: It is not necessary to assign a test value to link, since it is not used in our example in Figure 5.3

Table 5.1 MSC-TTCN-3 script mapping from Figure 5.3

Message Sequent Chart	TTCN-3 script
User-web site interactions	Signature Definition
Web page elements [URL, form name, 2 field names]	Record Type
Data value	Template

To sum up, we have illustrated the process of creating TTCN-3 test scripts from a use case scenario, which was expressed in MSC, namely:

- Identifying the interactions using TTCN-3 "signature".
- Defining information exchanged between user and Web site using TTCN-3 "record" type.
- Assigning test values in TTCN-3 "template".
- Creating test cases following the signature format (see Table 5.1).

We skip the detailed description of other TTCN-3 file definitions such as "module", "component", "port" and "control" as their definitions are quite similar to regular TTCN-3 test scripts. A complete list of TTCN-3 test scripts can be found in Appendix A.

5.4 Parser Design and Justification

To facilitate our approach, a parser that translates TTCN-3 to Java test code is introduced. In general, the parser works in three consecutive phases: the input file tokenizing phase, the data extraction phase and the output file construction phase.

First, the parser will read in the TTCN-3 file and chop it into an array of single words by previously defined tokens, for example a space, a tab or a new line. Next, we define TTCN-3 keywords such as "module", "record", "template", "testcase", etc. These key words in TTCN-3 will map a Java class, a variable type, a variable value and a test method respectively. For each and every one of the keywords the parser encounters, it extracts the value that follows this keyword and saves it to a Java collection, such as an "ArrayList" or a "HashMap". At last, the parser fetches those saved key/value pairs in the proper Java collection along with the HttpUnit API methods that process these keywords and their counterpart values to construct the output Java test file. To better understand the working mechanism of the parser, let us see an example. Suppose we have a TTCN-3 template definition as:

```
template myUrlType
loginurl:={"http://localhost:8000/petstore/"};
```

At the data extraction phase, the parser first recognizes the keyword "template" in the TTCN-3 file. A temple is supposed to map onto a variable value. The absolute position of this keyword is marked as an integer "i", so this keyword "template" could be represented as "word[i]" in the TTCN-3 file ArrayList (the TTCN-3 file is first chopped into single words and saved into a Java ArrayList collection). Figure 5.4 shows the example template definition.

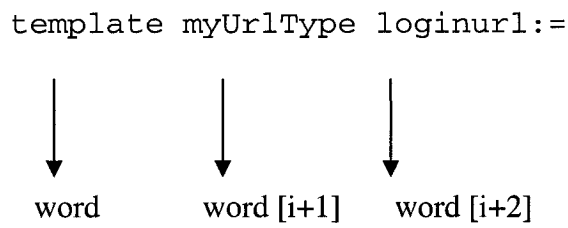

Figure 5.4 A Sample Template Definition

The consecutive words – "word [i+1]" is the "type" of this template, in this case myUrlType, and word [i+2] – "loginurl" is the "name" of this template. Similarly, words in the parenthesis are treated as the "value" of this template. The "type", "name" and "value" of this template are saved into different Java collections, in this case we setup a Java ArrayList whose elements are Java HashMaps.

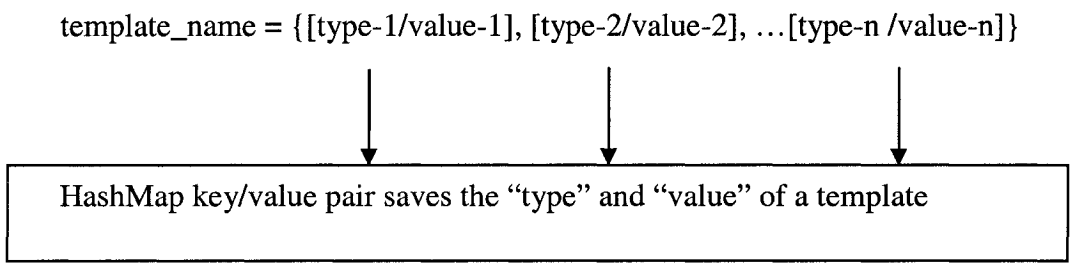

Figure 5.5 Template Storage Structure

Figure 5.5 depicts the template storage structure. The key of the HashMap is the template "type", and the value of the HashMap is the template "value". The "template_name" serves as the Java ArrayList element variable.

In the construction phase, we match template ArrayList's elements - template "names" with the testcase signature call's parameters and extract the proper HashMap key/value pair (See Figure 5.6).

```
if(word[i].equals("template")){
HashMap hm = new HashMap();
ArrayList alist = new ArrayList();
String temptype = word[i+1];
String tempname = word[i+2];
int fieldstartpos = findfirst(i, "{", word);
int fieldendpos = fef(fieldstartpos, word);
String tempvalue = content(fieldstartpos, fieldendpos, word);
System.out.println(tempvalue+" ");
alist.add(temptype);
alist.add(tempvalue);
```

Figure 5.6 Parser Code Fragment for Parsing Keyword "template"

After the data extraction phase, the parser begins to construct the output Java test file. TTCN-3 test cases send blocking-procedure calls to the SUT and wait for the reply. For example, we have a signature defined as:

```
signature testLoginCall(myUrlType url, myFormNameType
fName, myFormType uName,myFormType pwd, mySubmitType sub);
```

The calling signature parameters include a type and a value. The parser will identify the "type" of a signature parameter, for example, "myUrlType". This indicates the operations that are relevant to the URL process. Then, the parser will setup a "WebConversation" object in HttpUnit API and call the "getResponse()" method which returns a "WebResponse" object. The value of the parameter is assigned to the actual URL value in the test case call in Java. Figure 5.7 shows an example of processing "myUrlType".

```
if (key.equals("myUrlType")) {
//assign url value to the getResponse(url)method call
String url = value;
out.write("String url = "+url+";\n");
        out.write("WebLink link;\n");
        out.write("WebForm formname;\n");
        out.write("WebConversation wc = new WebConversation();\n");
        out.write("WebResponseresp = wc.getResponse(url);\n");
}
```

Figure 5.7 Code fragment for processing myUrlType to output Java file

The output Java code should be:

```
String url = "http://localhost:8000/petstore/";
WebLInk link;
WebForm formname;
WebConversation we = new WebConversaton();
WEbResponse resp = wc.getResponse(url);
```

We mapped TTCN-3 module to a Java Class name, a TTCN-3 testcase to a Java test method and a TTCN-3 template to a Java ArryList object. Here is the mapping list describing relationship between TTCN-3 and Java:

```
TTCN-3 module - Java class
TTCN-3 testcase - Java test method
TTCN-3 template - Java ArrayList object
```

When parsing a TTCN-3 "testcase" keyword, the parser identifies the signature call, which contains the TTCN-3 template names, and the "getreply" keyword whose

subsequent string contains the returned page title. This returned page title string will work as a verdict parameter in HttpUnit's assertEquals method.

When parsing a TTCN-3 "template" keyword, we use a Java ArrayList object to hold a key/value pair, whose key maps to a TTCN-3 "record type" name, while the ArrayList value maps to the TTCN-3 template value. TTCN-3 record types define distinct web page elements such as URL, form names, form filed names, links, submit buttons etc. We may define other Web page elements, such as tables and frames to facilitate different user interactions with a Web site. We can also add processing logic into our parser to parse these elements.

The parser is designed to work in three consecutive phases that process TTCN-3 files to Java test code and it is pattern-driven and keyword driven. The main drawback of this approach is that it is tool-dependent. If we need to use another open source tool, a new parser will have to be implemented. However, the tokenizer and the data extraction phase could be re-used and speed up the genteration of a parser for a new tool.

5.5 Summary

Modern e-commerce applications consist of complicated business logic and sophisticated software systems. The ever-increasing competition in the e-commerce arena requires increased revenue, quality of service with reduced costs and a reduced time-to-market. We need new software development methodologies to meet these challenges and supply solutions for such systems.

Extreme Programming (XP) breaks development into small size chunks and relies on daily face-to-face team communication and lots of automated testing. The assumption that specifications will change is built into the process. Projects are developed in increments, with just enough code written to implement the required functions, and everything (the design, the features, and the code) is constantly evaluated and adjusted to accomplish the desired result.

To meet the new challenges of e-commerce testing, we combine the advantages of an agile development methodology and formal method specification techniques to achieve the maximum benefit. This approach is facilitated by introducing a parser that translates TTCN-3 test scripts into Java test code, and then applying these Java test files to the selected open source test tool.

Once a complete TTCN-3 BNF grammar is available, one can use a standard parser-generator to generate the parser automatically. Our parser was developed manually, using patters. However, reworking the parser for a new tool should only affect the Java construction part. Thus, our approach of modifying the Java construction components only helps to speed up the generation of a parser for a different tool than HttpUnit.

Chapter 6 Case study

First, an overview and brief justification for the choice of case study components are presented. This is followed by detailed descriptions of the system under test (Sun Java Pet Store v.1.3.1) including key uses case, a brief description of the functional test tool (HttpUnit1.5.4) and a brief review of TTCN-3. Then, we proceed to derive test cases in TTCN-3 from the selected use cases to achieve FAST, FET, TOFT and Boundary Interior coverage. Next, we apply our parser to yield the Java code included HttpUnit invocations for these test cases. Then, we use Apache Ant 1.5 to build and execute the test code, and collect test results. Finally, we identify any additional test requirements and modify the TTCN-3 scripts accordingly.

6.1 Case Study Overview

Functions or features of an e-commerce system should be defined in system functional specifications, which act both as the contract between developers and stakeholders and the starting point for functional test design. Natural language descriptions of system specifications consist of many use cases that can be represented in MSC or other formal and unambiguous formats like UML State diagrams or Sequence diagrams. These diagrams should be shared between system developers and testers who implement test cases based on these diagrams and a set of best practice Web functional test design strategies [Nguyen03]. Test cases define test steps, test data and expected results. TTCN-3 test scripts are implemented by following these test case criteria.

The TTCN-3 to Java code parser translates TTCN-3 test scripts and generates Java test code using open source HttpUnit APIs. The generated Java test code simulates a Web client that interacts with the SUT – Sun Pet Store and generates test results. Apache Ant orchestrates the parsing and test execution process to facilitate XP's automatic testing and continuous integration practices. Figure 6.1 shows a road map of our case study. We will discuss each of these steps in detail in the upcoming sections.

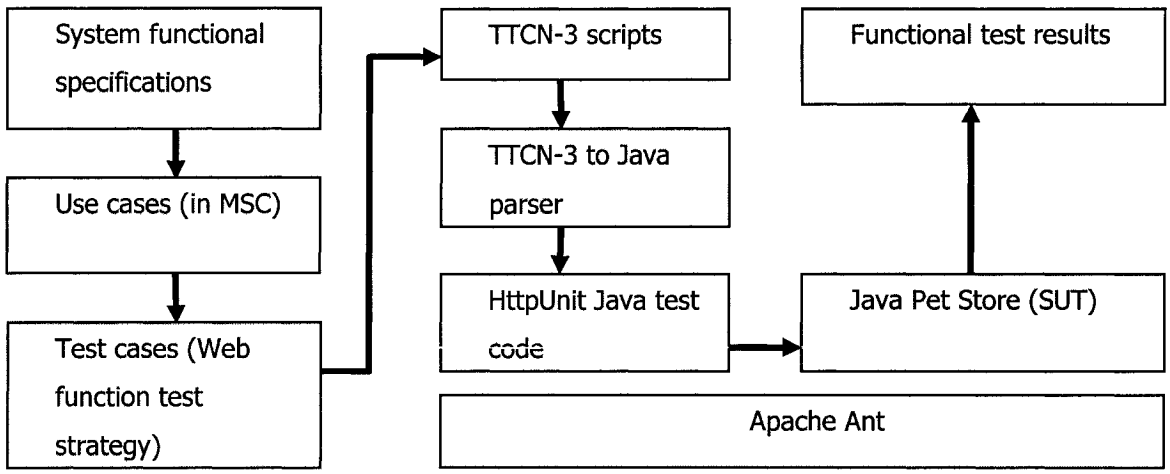

Figure 6.1 Case Study Overview

6.2 Case Study Selection Rationale and SUT Functional Specifications

6.2.1 SUT, Test Tool and Test Script Language Selection Rationale

We chose Sun's Java Pet Store 1.3.1 [Sun Pet Store] as our system under test. This was a good choice because Pet store is a realistic, though not suitable for commercial use sample Java e-commerce application based on J2EE infrastructure. It was developed by Sun's Blueprint project [Blueprints] to help Java developers learn and experiment with applications built on the Java 2 platform. Blueprint provides complete guidelines, patterns and code to showcase the capability of the J2EE platform to develop flexible, scalable, cross-platform enterprise applications.

For the functional test tool, we chose HttpUnit for the following reasons. First, it is an open source test tool with a useful and easy to use API. There is no license fee overhead. Secondly, to take advantage of XP's core practices, especially automatic testing and continuous integration, functional test tools need to be small and effective, as developers

may be discouraged by lengthy integration test executions on a regular basis. As we have discussed in Chapter 5, HttpUnit is designed to test end-to-end Web based Java applications. Also, with a small but practical API set, HttpUnit test cases could be integrated into the Java project development process, thus enabling the automatic build tool to integrate and test the whole application continuously.

We chose TTCN-3 as the functional test script language. One of the reasons was TTCN-3 is the international standard test language. We have learned in chapter 4 that TTCN-3 offers three presentation formats- textual core format, tabular format and graphical MSC format. In this case study, we first specify selected use cases in MSC as it gives a clear and intuitive view of the test behavior. Then we derive the textual core format TTCN-3 test cases based on these previously defined MSCs, since the textual TTCN-3 specification is easier to translate into other languages such as Java. Specifically, we have automated this last step by developing a translator from TTCN-3 to Java.

6.2.2 Background: SUT Functional Specifications

Sun's Pet Store 1.3.1 simulates an online pet store enterprise that sells animals to customers. The sample application comprises four separate sub-applications that cooperate to fulfill the enterprise's business needs, each of which is a J2EE application (see Figure 6.2):

- Pet store e-commerce Web site - a Web application which shoppers use to purchase merchandise through a Web browser
- Pet store administration application - a Web application that enterprise administrators use to view sales statistics and manually accept or reject orders.
- Order processing center (OPC) - a process-oriented application that manages order fulfillment by providing the services to other enterprise participants
- Supplier - a process-oriented application that manages shipping products to customers by providing services such as receiving order from OPC and maintaining inventory database

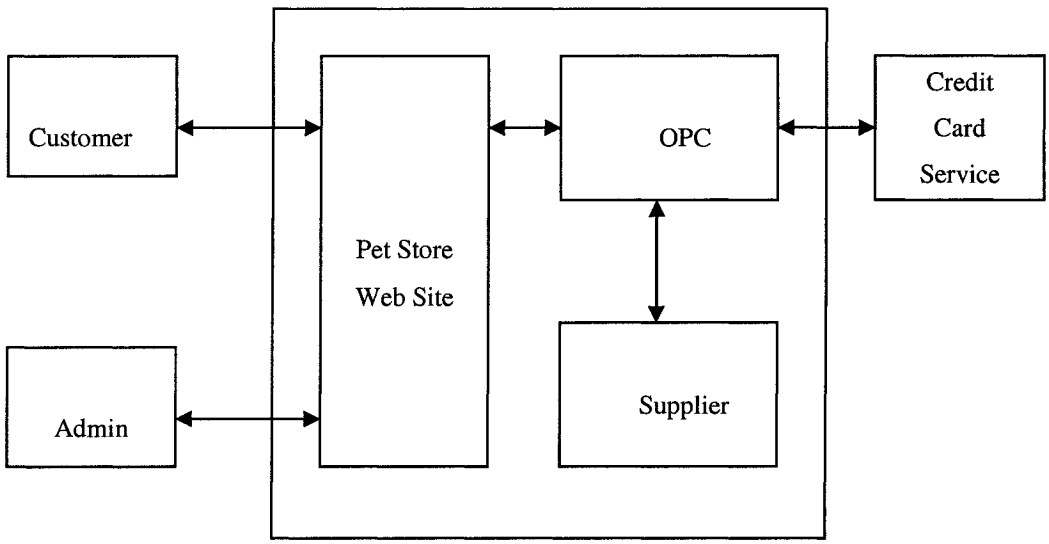

Figure 6.2 Pet Store Architecture

The Pet Store Web site is the part of the sample application that provides customers with online shopping. Through a Web browser, a customer can browse the catalog, place items to purchase into a virtual shopping cart, create and sign in to a user account, and purchase the shopping cart contents by placing an order with a credit card. This section presents a look at the Web site from the user's point of view. Figure 6.3 shows the use cases between the customer and the pet store Web site. In this section, we focus on three use cases namely "user registration", "user login" and "place order", and translate them into MSCs to set the stage for deriving TTCN-3 test scripts.

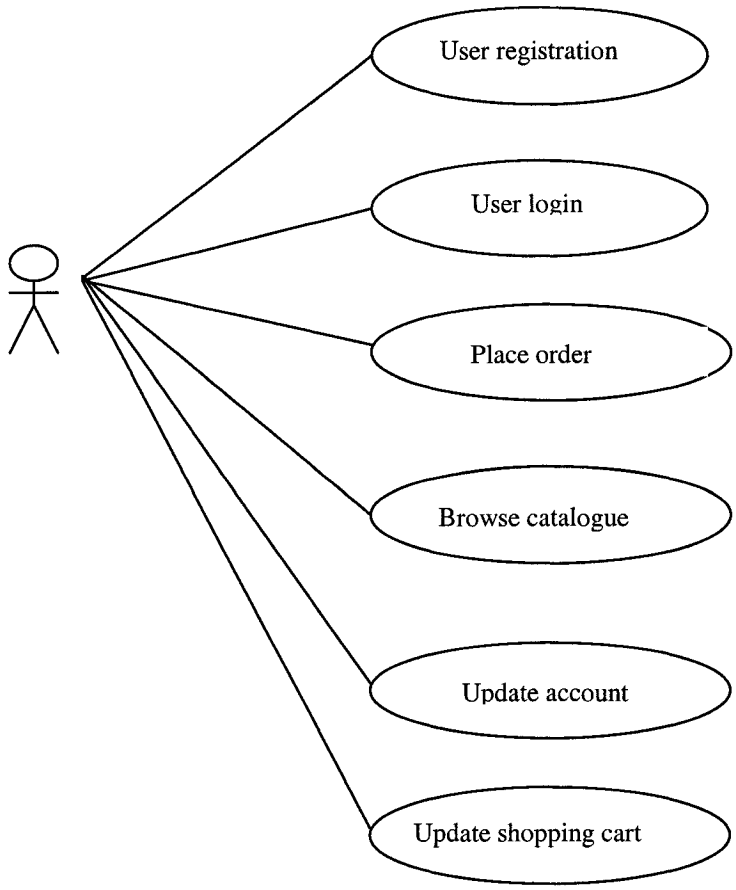

Figure 6.3 Use Cases Between Customer and Web Site

Customer Registration Use Case

This use case is used in the Boundary test. The registration use case allows a customer to register into the system by providing a valid user name and a password. A logged in user may browse or edit the account information and update it by submitting the new personal information form. Figure 6.4 shows customer login and new customer registration forms.

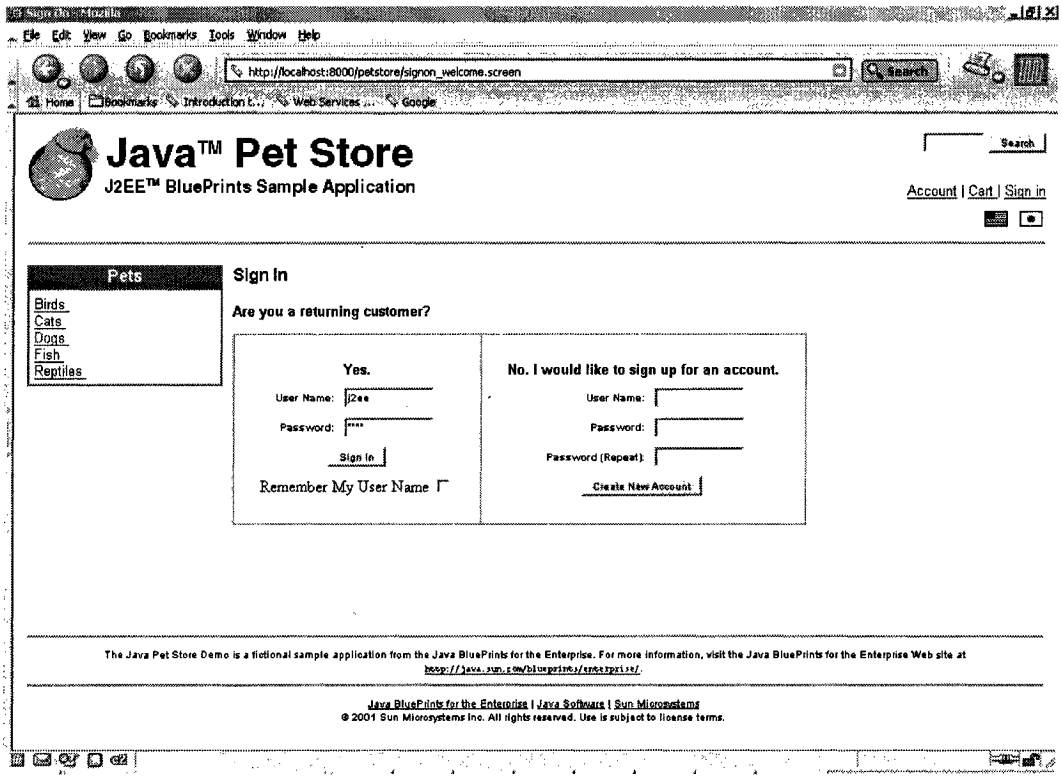

Figure 6.4 Customer Login Form and Registration Form

If a customer visits the store for the first time, she/he needs to register a new user account before making a purchase by following these steps:

1. Send login page URL
2. Fill out the form titled "I would like to sign up for an account", and click "Create New Account".
3. Enter account information and click "Submit".
4. Web site returns Welcome page or registration error page
5. The customer registration use case is presented in MSC in Figure 6.5

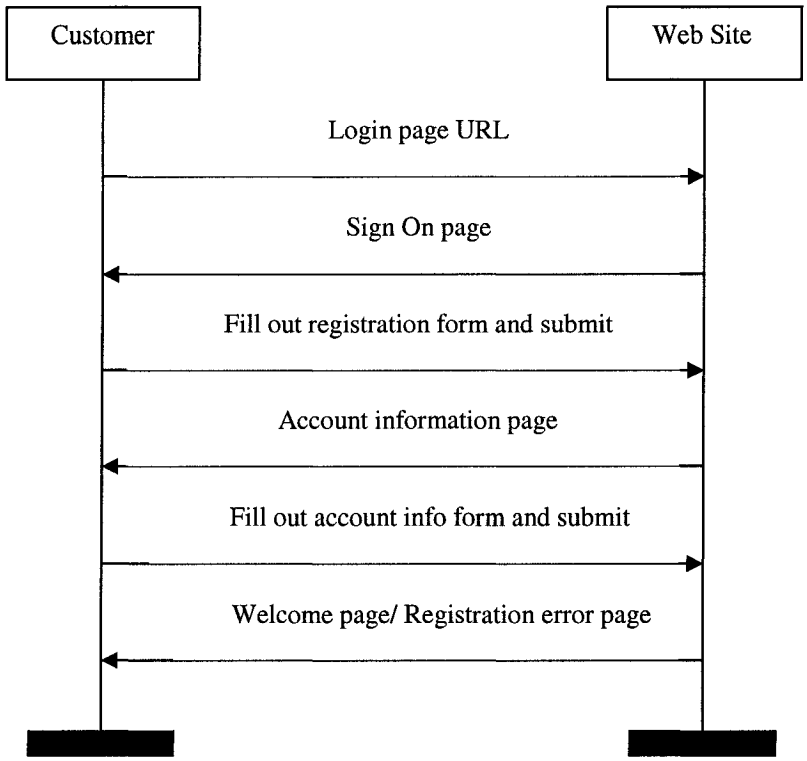

Figure 6.5 Customer Registrations Use Case MSC

Customer Login Use Case

This use case can be applied to many test design strategies including FET and FAST tests in the next section. The customer login use case allows a customer login with the correct username/password combination. If the correct username and password are submitted, the Web site returns a "Welcome" page enabling the user to perform subsequent online shopping, if the username/password pair is not valid, the Web site returns a "Sign On Error" page. Figure 6.6 shows this use case in MSC.

1. Send login page URL.
2. Fill out the form titled "Yes.", and click "Sign In".
3. Web site will return a "Welcome" page or a "Sign On Error" page.

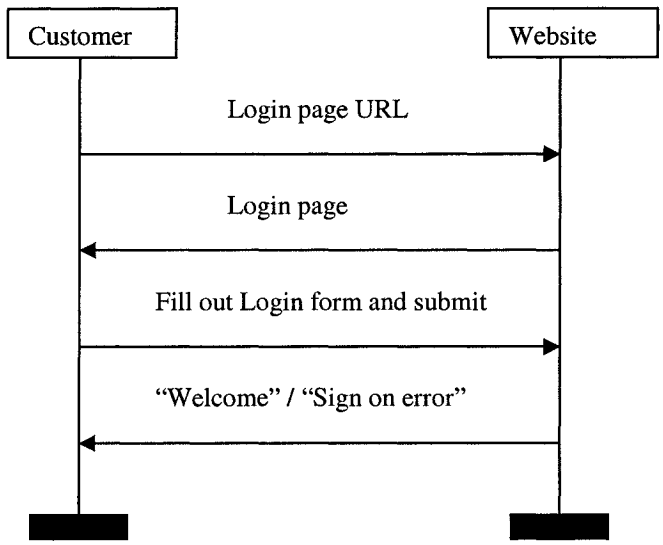

Figure 6.6 Customer Login Use Case MSC

Note, checking the returned page title is a simple and effective way to validate that the expected web page is returned. Figure 6.7 shows a web page with the title "Welcome" after customer successfully logged in.

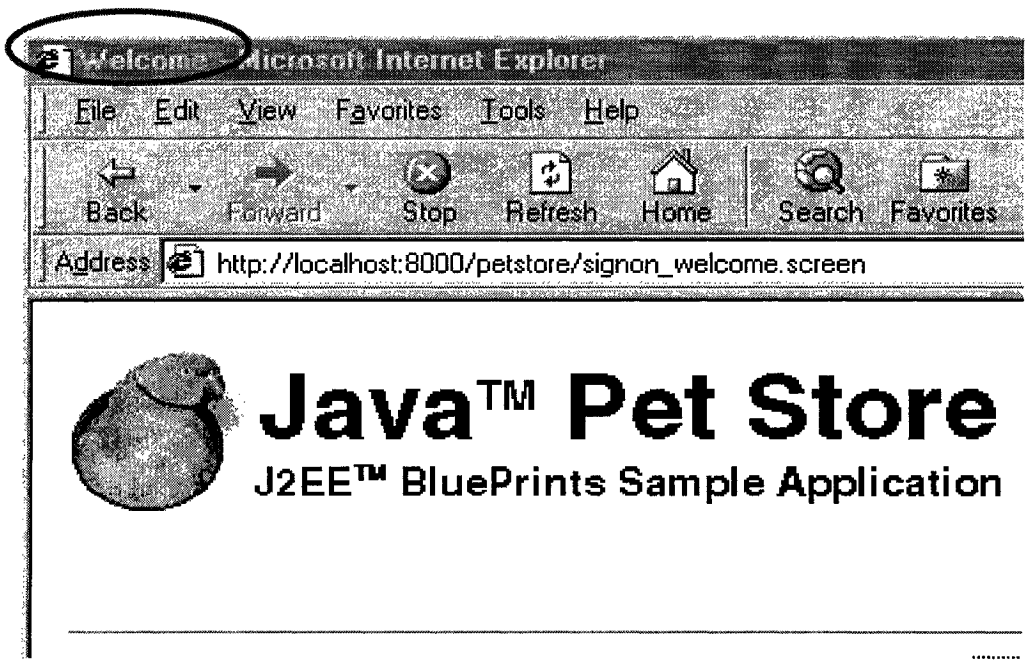

Figure 6.7 Returned Page Title – Welcome String

Customer Place Order Use Case

This use case is used in TOFT test. To purchase a bulldog, for example, a customer follows these steps:

1. Send login page URL.
2. Fill out Login page and submit.
3. On the Welcome page, Click "Dogs" link.
4. Click "Bulldog" link.
5. Click "Add to Cart" link in the row for the "Male Adult Bulldog".
6. Click "Check Out". (A form for your billing and shipping info appears. For your convenience, it's already filled out.)
7. Click "Submit".
8. Web site returns "Order Placed" page.

Figure 6.8 shows this use case in MSC.

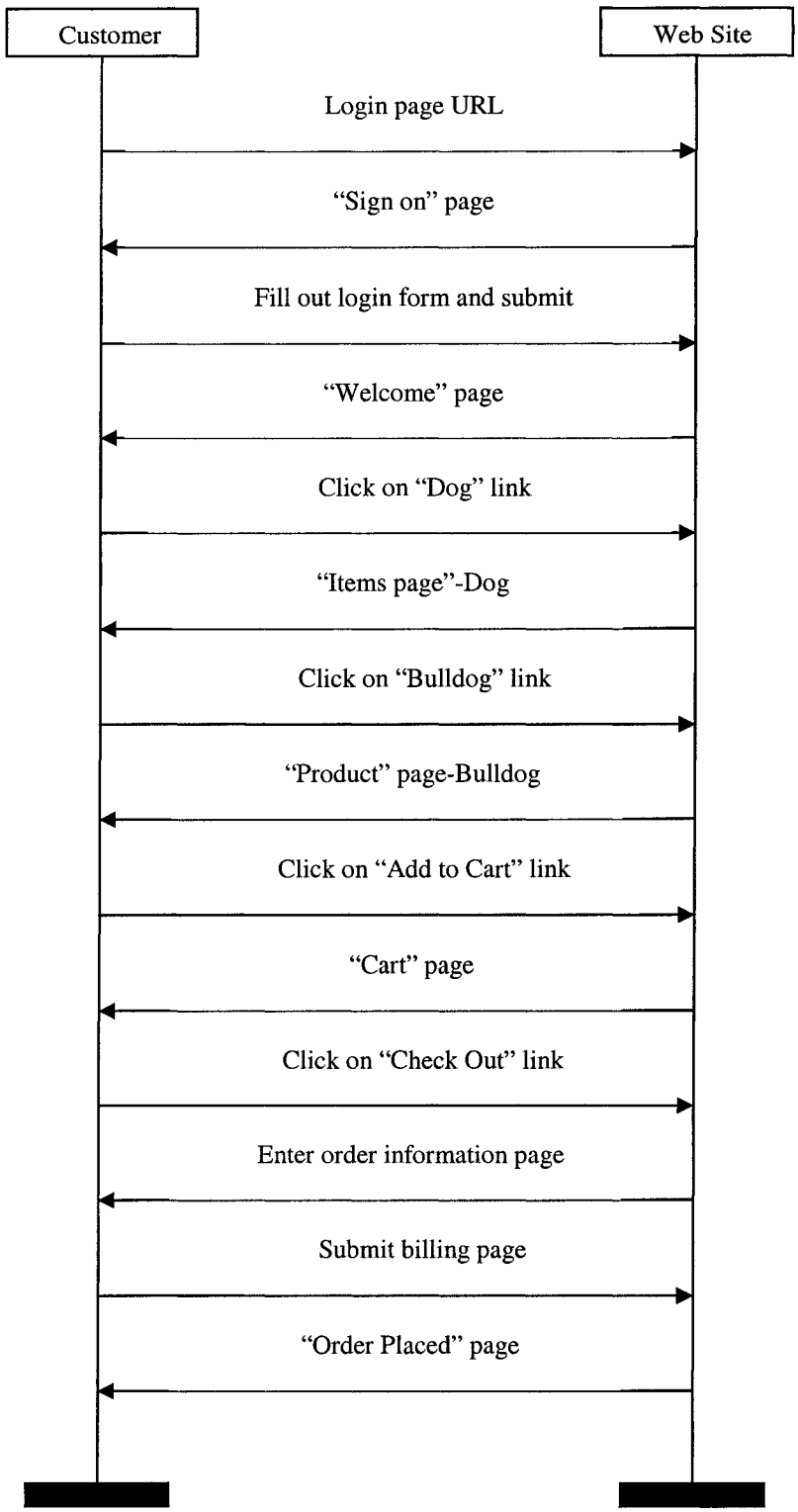

Figure 6.8 Customer Place Order Use Case MSC

There are only three of the many use cases needed to complete the specification of the SUT. More use cases appeared at Java Pet store documents [Sun Pet Store]. The purpose of this section was to illustrate the close relationship between each use case and a corresponding MSC (alternatively, UML sequence diagram).

6.3 Derivation Strategy of TTCN-3 Test Scripts from Use Cases

As we discussed at the beginning of this chapter, Web-based e-commerce system functional test cases are implemented by following system specifications and a set of best practice Web functional test case design strategies. In section 6.2 we gave a walk through description of the SUT from the functional perspective, "customer registration", "customer login" and "customer place order" are three important use cases we will apply to implement test cases in TTCN-3. Also, by following the discussion in section 3.8 "Functional Test Case Design Strategy for Web Applications" we implemented the following test cases in TTCN-3, see table 6.1.

Table 6.1 Conducted tests in TTCN-3

Type of Test	Use Case	Test Case
FAST	Login [Figure 6.6]	testcorrectlogin
FAST	Click link	testjavasoftlink
FAST	Click link	testsunlink
FET	Login [Figure 6.6]	testblankusernamelogin
FET	Login [Figure 6.6]	testblankpasswordlogin
FET	Login [Figure 6.6]	testwronglogin
TOFT	Place order [figure 6.8]	testplaceorder
Boundary	Registration [Figure 6.5]	test1reg
Boundary	Registration [Figure 6.5]	test2reg
Boundary	Registration [Figure 6.5]	test24reg
Boundary	Registration [Figure 6.5]	test25reg
Boundary	Registration [Figure 6.5]	test26reg

6.3.1 FAST Test Case

For the screen shot in Figure 6.4, we select 3 example test cases: a form test (login form) and two link tests (Java Software link and Sun link) to facilitate this test design. FAST tests are shown in Test Case 6.1 to 6.3.

Test Case 6.1 FAST Test: testcorrectlogin

Test case name: testcorrectlogin
Test step: see Figure 6.6
Test data:
URL: http://localhost:8000/petstroe/signon_welcome.screen
Form name: existingcustomer
User name field name: j_username
User name field value: wei
Password field name: j_password
Password field value: weipass
Form action: submit
Expected result: return page entitled "Welcome"

Test Case 6.2 FAST Test: testjavasoftlink

Test case name: testjavasoftlink
Test step: go to Pet Store home page and click on "Java Soft" link
Test data:
URL: http://localhost:8000/petstroe/signon_welcome.screen
Link name: Java Soft
Expected result: return page entitled "The Source for Java Technology"

Test Case 6.3 FAST Test: testsunlink

Test case name: testsunlink
Test step: go to Pet Store home page and click on "Sun Microsystems" link
Test data:
URL: http://localhost:8000/petstroe/signon_welcome.screen

Link name: Sun Microsystems
Expected result: return page entitled "Sun Microsystems"

From the functional use case specification described in MSC (see Figure 6.6) and above FAST test case specifications we derived the following TTCN-3 test script including record, template, signature definitions and 3 FAST test cases.

```
//TTCN-3 module is named by the test strategy
module FAST{

//skip over port and component definitions …
//type record definitions define variable types
type record myUrlType{charstring url};
type record myFormNameType{charstring formname};
type record myFormType{charstring field};
type record myLinkType{charstring link};
type record mySubmitType{charstring submit};

//templates define actual test data
template myUrlType
loginurl:={"http://localhost:8000/petstore/signon_welcome.screen"};
template myFormNameType loginformname:={"existingcustomer"};

template myFormType username:={"j_username", "wei"};
template myFormType password:={"j_password", "weipass"};
template mySubmitType submit:={"submit"};

template myLinkType javasoftlink:={"Java_ Software"};
template myLinkType sunlink:={"Sun_Microsystems"};

signature testLoginCall{myUrlType url, myFormNameType fName,
myFormType uName,myFormType pwd, mySubmitType sub};
signature testLinkCall{myUrlType url, myLinkType link};

//testcases define test interactions and expected result
testcase testcorrectlogin runs on MTCType{
```

```
  var PTCType formptc;
  formptc.Create;
  connect(formptc:pco, system:tsipco);
  map(formptc:cp, mtc:cp);
  pco.call(testLoginCall:{loginurl,loginformname, username,
password,submit})
  alt{
  []pco.getreply("Welcome"){
      verdict.set(pass);
      }
    []pco.catch{
      verdict.set(fail);
      }
  stop;
  }
  }

  testcase testjavasoftlink runs on MTCType{
  var PTCType linkptc;
  linkptc.Create;
  connect(linkptc:pco, system:tsipco);
  map(linkptc:cp, mtc:cp);
  pco.call(testLinkCall:{loginurl, javasoftlink})
  alt{
  []pco.getreply("The_Source_for_Java_Technology"){
      verdict.set(pass);
      }
    []pco.catch{
      verdict.set(fail);
      }
  stop;
  }
  }

  testcase testsunlink runs on MTCType{
  var PTCType linkptc;
  linkptc.Create;
  connect(linkptc:pco, system:tsipco);
```

```
map(linkptc:cp, mtc:cp);
pco.call(testLinkCall:{loginurl, sunlink})
alt{
[]pco.getreply("_Sun_Microsystems_"){
    verdict.set(pass);
    }
  []pco.catch{
    verdict.set(fail);
    }
stop;
}
}
}
```

In practice, many more FAST test cases would be added, organized by customer requirements and screen.

6.3.2 FET Test Case

According to the customer login specification, user name and password should be registered before entering the Pet Store system. In the first test case, we deliberately enter a non-registered user/password pair. The second test case leaves the user name field blank and the third test case leaves the password field blank to see if the system could return an error page in these forced error tests. FET tests are shown in Test Case 6.4 to 6.6.

Test Case 6.4 FET Test: testblankusernamelogin

Test case name: testblankusernamelogin
Test step: see Figure 6.6
Test data:
URL: http://localhost:8000/petstroe/signon_welcome.screen
Form name: existingcustomer
User name field name: j_username
User name field value: “”

Password field name: j_password
Password field value: password
Form action: submit
Expected result: return page entitled "Sign On Error"
Actual result: page entitled "Sign On Error" returned
Status: Pass

Test Case 6.5 FET Test: testblankpasswordlogin

Test case name: testblankpasswordlogin
Test step: see Figure 6.6
Test data:
URL: http://localhost:8000/petstroe/signon_welcome.screen
Form name: existingcustomer
User name field name: j_username
User name field value: "wei"
Password field name: j_password
Password field value: ""
Form action: submit
Expected result: return page entitled "Sign On Error"
Actual result: page entitled "Sign On Error" returned
Status: Pass

Test Case 6.6 FET Test: testwronglogin

Test case name: testwronglogin
Test step: see Figure 6.6
Test data:
URL: http://localhost:8000/petstroe/signon_welcome.screen
Form name: existingcustomer
User name field name: j_username
User name field value: "notwei"
Password field name: j_password
Password field value: "notweipass"

Form action: submit
Expected result: return page entitled "Sign On Error"
Actual result: page entitled "Sign On Error" returned
Status: Pass

From the functional use case specification described in MSC (see Figure 6.6) and above FET test case specifications we derived the following TTCN-3 test script including record, template, signature definitions and 3 FET test cases.

```
//TTCN-3 module is named by the test strategy
module FET{

//skip over port and component definitions…

//type record definitions define variable types
type record myUrlType{charstring url};
type record myFormNameType{charstring formname};
type record myFormType{charstring field};
type record myLinkType{charstring link};
type record mySubmitType{charstring submit}

//templates define actual test data
template myUrlType
loginurl:={"http://localhost:8000/petstore/signon_welcome.screen"};
template myFormNameType loginformname:={"existingcustomer"};

template myFormType blankusername:={"j_username", ""};
template myFormType password:={"j_password", "password"};

template myFormType username:={"j_username", "wei"};
template myFormType blankpassword:={"j_password", ""};

template myFormType wrongusername:={"j_username", "notwei"};
template myFormType wrongpassword:={"j_password", "notweipass"};
template mySubmitType submit:={"submit"};
```

```
  signature testLoginCall{myUrlType url, myFormNameType fName,
myFormType uName,myFormType pwd, mySubmitType sub};

  //testcases define test interactions and expected result
  testcase testblankusernamelogin runs on MTCType{
  var PTCType formptc;
  formptc.Create;
  connect(formptc:pco, system:tsipco);
  map(formptc:cp, mtc:cp);
  pco.call(testLoginCall:{loginurl,loginformname, blankusername,
password,submit})
  alt{
  []pco.getreply("Sign_On_Error"){
      verdict.set(pass);
      }
    []pco.catch{
      verdict.set(fail);
      }
  stop;
  }
  }

  testcase testblankpasswordlogin runs on MTCType{
  var PTCType formptc;
  formptc.Create;
  connect(formptc:pco, system:tsipco);
  map(formptc:cp, mtc:cp);
  pco.call(testLoginCall:{loginurl,loginformname,  username,
blankpassword,submit})
  alt{
  []pco.getreply("Sign_On_Error"){
      verdict.set(pass);
      }
    []pco.catch{
      verdict.set(fail);
      }
  stop;
  }
```

```
}

testcase testwronglogin runs on MTCType{
var PTCType formptc;
formptc.Create;
connect(formptc:pco, system:tsipco);
map(formptc:cp, mtc:cp);
pco.call(testLoginCall:{loginurl, loginformname, wrongusername,
wrongpassword, submit})
alt{
  []pco.getreply("Sign_On_Error"){
    verdict.set(pass);
    }
  []pco.catch{
    verdict.set(fail);
    }
stop;
}
}
}
```

6.3.3 TOFT Test Case

TOFT tests are designed to test features or functions of the system, we choose "Customer Place Order" use case because it includes several steps to achieve the complete customer place order function. TOFT test is shown in Test Case 6.7.

A complete place order function includes: user correctly login, user browses "Item" catalog, user browses "Product", user adds product in the shopping cart, user checks out and user receives an "Order Placed" confirmation web page. We simulate a user buying a Persian cat in this test case.

Test Case 6.7 TOFT Test: testplaceorder

Test case name: testplaceorder
Test step: see Figure 6.8
Test data:

URL: http://localhost:8000/petstroe/signon_welcome.screen
Form name: existingcustomer
User name field name: j_username
User name field value: "wei"
Password field name: j_password
Password field value: "weipass"
Form action: submit
Items link: "Cats"
Product link: "Persian"
Add to cart link: "Add to Cart"
Check out link: "Check Out"
Billing form name: "order ino"
Billing form action: submit
Expected result: return page entitled "Order Placed"
Actual result: page entitled "Order Placed" returned
Status: Pass

From the functional use case specification described in MSC (see Figure 6.8) and the above TOFT test case specification we derived the following TTCN-3 test script including record, template, signature definitions and the TOFT test case.

```
//TTCN-3 module is named by the test strategy
module TOFT{

//skip over port and component definitions…

//type record definitions define variable types
type record myUrlType{charstring url};
type record myFormNameType{charstring formname};
type record myFormType{charstring field};
type record myLinkType{charstring link};
type record mySubmitType{charstring submit}

//templates define actual test data
```

```
  template myUrlType
loginurl:={"http://localhost:8000/petstore/signon_welcome.screen"};
  template myFormNameType loginformname:={"existingcustomer"};
  template myFormType correctusername:={"j_username", "wei"};
  template myFormType correctpassword:={"j_password", "weipass"};
  template mySubmitType submit:={"submit"};
  template myLinkType catlink:={"Cats"};
  template myLinkType persianlink:={"Persian"};
  template myLinkType addtocartlink:={"Add_to_Cart"};
  template myLinkType checkoutlink:={"Check_Out"};
  template myFormNameType submitformname:={"order_ino"};

  signature testOrderCall(myUrlType url, myLinkType link0,
myFormnameType,fName, mySubmitType sub, myLinkType link1, myLinkType
link2, myLinkType link3, myLinkType link4);

  //testcases define test interactions and expected result
  testcase testplaceorder runs on MTCType{
  var PTCType placeorderptc;
  placeorderptc.Create;
  connect(placeorderptc:pco, system:tsipco);
  map(placeorderptc:cp, mtc:cp);
  pco.call(testOrderCall:{orderurl,signinlink, loginformname,
correctusername, correctpassword, submit, catlink, persianlink,
addtocartlink, checkoutlink, submitformname, submit})
  alt{
    []pco.getreply("Order_Placed"){
      verdict.set(pass);
      }
    []pco.catch{
      verdict.set(fail);
      }
  stop;
  }
  }
  }
```

6.3.4 Boundary Tests

We use the "customer registration form" test as an example to illustrate this test design. The requirement for user name length should be 1-25 characters. The boundary is around 0, 1, 2, 24, 25 and 26. To demonstrate the feasibility of our approach, we skip the 0 length user name (blank user name field) test case here and then add this test case later in section 6.6. Boundary tests are shown in Test Case 6.8 to 6.12 where we set 1, 2, 24, 25, 26 character length user names.

Test Case 6.8 Boundary Test: test1reg

Test case name: test1reg
Test step: see Figure 6.5
Test data:
URL: http://localhost:8000/petstroe/signon_welcome.screen
Form name: newcustomer
User name field name: j_username
User name field value: b
Password field name: j_password
Password field value: password
Form action: submit
Expected result: return page entitled "Create Customer"
Actual result: page entitled "Create Customer" returned
Status: Pass

Test Case 6.9 Boundary Test: test2reg

Test case name: test2reg
Test step: see Figure 6.5
Test data:
URL: http://localhost:8000/petstroe/signon_welcome.screen
Form name: newcustomer
User name field name: j_username
User name field value: b2
Password field name: j_password

Password field value: password
Form action: submit
Expected result: return page entitled "Create Customer"
Actual result: page entitled "Create Customer" returned
Status: Pass

Test Case 6.10 Boundary Test: test24reg

Test case name: test24reg
Test step: see Figure 6.5
Test data:
URL: http://localhost:8000/petstroe/signon_welcome.screen
Form name: newcustomer
User name field name: j_username
User name field value: b111111111111111111111124
Password field name: j_password
Password field value: password
Form action: submit
Expected result: return page entitled "Create Customer"
Actual result: page entitled "Create Customer" returned
Status: Pass

Test Case 6.11 Boundary Test: test25reg

Test case name: test25reg
Test step: see Figure 6.5
Test data:
URL: http://localhost:8000/petstroe/signon_welcome.screen
Form name: newcustomer
User name field name: j_username
User name field value: b1111111111111111111111125
Password field name: j_password
Password field value: password
Form action: submit
Expected result: return page entitled "Create Customer"

Actual result: page entitled "Create Customer" returned
Status: Pass

Test Case 6.12 Boundary Test: test26reg

Test case name: test26reg
Test step: see Figure 6.5
Test data:
URL: http://localhost:8000/petstroe/signon_welcome.screen
Form name: newcustomer
User name field name: j_username
User name field value: b11111111111111111111111126
Password field name: j_password
Password field value: password
Form action: submit
Expected result: return page entitled "Duplicate Account"
Actual result: page entitled "Duplicate Account" returned
Status: Pass

From the functional use case specification described in MSC (see Figure 6.5) and above Boundary test case specifications we derived the following TTCN-3 test script including necessary record, template, signature definitions and 5 Boundary test cases.

```
//TTCN-3 module is named by the test strategy
module Boundary{

//skip over port and component definitions…

//type record definitions define variable types
type record myUrlType{charstring url};
type record myFormNameType{charstring formname};
type record myFormType{charstring field};
type record myLinkType{charstring link};
type record mySubmitType{charstring submit}
```

```
   //templates define actual test data
   template myUrlType
 regurl:={"http://localhost:8000/petstore/signon_welcome.screen"};
   template myFormNameType regformname:={"newcustomer"};

   template myFormType  username1:={"j_username", "b"};
   template myFormType  password1:={"j_password", "password"};

   template myFormType  username2:={"j_username", "b2"};
   template myFormType  password2:={"j_password", "password"};

   template myFormType  username24:={"j_username",
 "b111111111111111111111124"};
   template myFormType  password24:={"j_password", "password"};

   template myFormType  username25:={"j_username",
 "b111111111111111111111125"};
   template myFormType  password25:={"j_password", "password"};

   template myFormType  username26:={"j_username",
 "b111111111111111111111126"};
   template myFormType  password26:={"j_password", "password"};

   template mySubmitType submit:={"submit"};

   signature testRegCall{myUrlType url, myFormNameType fName,
myFormType uName,myFormType pwd, myFormType pwd2, mySubmitType sub};

   //testcases define test interactions and expected result

   testcase test1reg runs on MTCType{
   var PTCType formptc;
   formptc.Create;
   connect(formptc:pco, system:tsipco);
   map(formptc:cp, mtc:cp);
   pco.call(testRegCall:{regurl, regformname,  username1,  password1,
password-2, submit})
   alt{
```

```
  []pco.getreply("Create_Customer"){
      verdict.set(pass);
      }
    []pco.catch{
      verdict.set(fail);
      }
  stop;
  }
  }

  testcase test2reg runs on MTCType{
  var PTCType formptc;
  formptc.Create;
  connect(formptc:pco, system:tsipco);
  map(formptc:cp, mtc:cp);
  pco.call(testRegCall:{regurl,regformname, username2,
password2,password-2,submit})
  alt{
  []pco.getreply("Create_Customer"){
      verdict.set(pass);
      }
    []pco.catch{
      verdict.set(fail);
      }
  stop;
  }
  }

  testcase test24reg runs on MTCType{
  var PTCType formptc;
  formptc.Create;
  connect(formptc:pco, system:tsipco);
  map(formptc:cp, mtc:cp);
  pco.call(testRegCall:{regurl,regformname, username24,
password24,password-2,submit})
  alt{
  []pco.getreply("Create_Customer"){
      verdict.set(pass);
```

```
        }
      []pco.catch{
        verdict.set(fail);
        }
   stop;
   }
   }

   testcase test25reg runs on MTCType{
   var PTCType formptc;
   formptc.Create;
   connect(formptc:pco, system:tsipco);
   map(formptc:cp, mtc:cp);
   pco.call(testRegCall:{regurl,regformname, username25,
password25,password-2,submit})
   alt{
   []pco.getreply("Create_Customer"){
       verdict.set(pass);
       }
     []pco.catch{
       verdict.set(fail);
       }
   stop;
   }
   }

   testcase test26reg runs on{
   var PTCType formptc;
   formptc.Create;
   connect(formptc:pco, system:tsipco);
   map(formptc:cp, mtc:cp);
   pco.call(testRegCall:{regurl, regformname, username26, password26,
password-2,submit})
   alt{
     []pco.getreply("Duplicate_Account"){
       verdict.set(pass);
       }
     []pco.catch{
```

```
        verdict.set(fail);
        }
    stop;
    }
    }
    }
```

6.4 Applying the FTEXT Parser to Yield Java Test Code

We explained the working mechanism of the TTCN-3 to Java parser in the previous chapter. Every test case in the TTCN-3 script is parsed to a Java test method in the parsed Java test file. We skip over other parsed Java code but parsed Java methods to show that the parser achieved expected results.

6.4.1 FAST

FAST contains 3 test methods: testcorrectlogin(), testjavasoftlink() and testsunlink():

```
    public void testcorrectlogin() throws Exception{
    String url =
  "http://localhost:8000/petstore/signon_welcome.screen";
    WebLink link;
    WebForm formname;
    WebConversation wc = new WebConversation();
    WebResponse resp = wc.getResponse(url);
    formname = resp.getFormWithName("existingcustomer");
    formname.setParameter("j_username","wei");
    formname.setParameter("j_password","weipass");
    resp = formname.submit();
    System.out.println("Got Page "+resp.getTitle());
    String returnedtitle = resp.getTitle();
    assertEquals("Welcome", returnedtitle);
    }

    public void testjavasoftlink() throws Exception{
```

```
    String url =
  "http://localhost:8000/petstore/signon_welcome.screen";
    WebLink link;
    WebForm formname;
    WebConversation wc = new WebConversation();
    WebResponse resp = wc.getResponse(url);
    link = resp.getLinkWith("Java Software");
    resp = link.click();
    System.out.println("Got Page "+resp.getTitle());
    String returnedtitle = resp.getTitle();
    assertEquals("The Source for Java Technology", returnedtitle);
    }

    public void testsunlink() throws Exception{
    String url =
  "http://localhost:8000/petstore/signon_welcome.screen";
    WebLink link;
    WebForm formname;
    WebConversation wc = new WebConversation();
    WebResponse resp = wc.getResponse(url);
    link = resp.getLinkWith("Sun Microsystems");
    resp = link.click();
    System.out.println("Got Page "+resp.getTitle());
    String returnedtitle = resp.getTitle();
    assertEquals(" Sun Microsystems ", returnedtitle);
    }
```

From the result Java code snippet, we can see TTCN-3 testcase names are parsed to Java method names, for example:

```
    public void testcorrectlogin() throws Exception,
```

TTCN-3 template values are parsed to relevant Java variable values such as:

```
    String url =
  "http://localhost:8000/petstore/signon_welcome.screen";
    formname = resp.getFormWithName("existingcustomer");
```

The parser also generates the stub code for HttpUnit tests. For example, creates a new WebConversation object to simulate the behavior of a browser, inserts verdict code and prints to screen the returned page title:

```
WebConversation wc = new WebConversation();
…
System.out.println("Got Page "+resp.getTitle());
String returnedtitle = resp.getTitle();
assertEquals("Welcome", returnedtitle);
```

6.4.2 FET

FET contains 3 test methods: testblankusernamelogin(), testblankpasswordlogin() and testwronglogin():

```
public void testblankusernamelogin() throws Exception{
String url =
"http://localhost:8000/petstore/signon_welcome.screen";
WebLink link;
WebForm formname;
WebConversation wc = new WebConversation();
WebResponse resp = wc.getResponse(url);
formname = resp.getFormWithName("existingcustomer");
formname.setParameter("j_username","");
formname.setParameter("j_password","password");
resp = formname.submit();
System.out.println("Got Page "+resp.getTitle());
String returnedtitle = resp.getTitle();
assertEquals("Sign On", returnedtitle);
}

public void testblankpasswordlogin() throws Exception{
String url =
"http://localhost:8000/petstore/signon_welcome.screen";
WebLink link;
WebForm formname;
WebConversation wc = new WebConversation();
WebResponse resp = wc.getResponse(url);
```

```
    formname = resp.getFormWithName("existingcustomer");
    formname.setParameter("j_username","wei");
    formname.setParameter("j_password","");
    resp = formname.submit();
    System.out.println("Got Page "+resp.getTitle());
    String returnedtitle = resp.getTitle();
    assertEquals("Sign On", returnedtitle);
    }

    public void testwronglogin() throws Exception{
    String url =
  "http://localhost:8000/petstore/signon_welcome.screen";
    WebLink link;
    WebForm formname;
    WebConversation wc = new WebConversation();
    WebResponse resp = wc.getResponse(url);
    formname = resp.getFormWithName("existingcustomer");
    formname.setParameter("j_username","notwei");
    formname.setParameter("j_password","notweipass");
    resp = formname.submit();
    System.out.println("Got Page "+resp.getTitle());
    String returnedtitle = resp.getTitle();
    assertEquals("Sign On Error", returnedtitle);
    }
```

A non-registered user name and password pair "notwei" and "notweipass" is selected for the third test case. These test data values are parsed and set into HttpUnit test forms. Similarly, a blank space is set to the user name field and password field in the first and second test case respectively.

6.4.3 TOFT

TOFT contains one test method: testplaceorder():

```
    public void testplaceorder() throws Exception{
    String url =
  "http://localhost:8000/petstore/signon_welcome.screen";
    WebLink link;
```

```
WebForm formname;
WebConversation wc = new WebConversation();
WebResponse resp = wc.getResponse(url);
link = resp.getLinkWith("Sign in");
resp = link.click();
System.out.println("Got Page " + resp.getTitle());
formname = resp.getFormWithName("existingcustomer");
formname.setParameter("j_username","wei");
formname.setParameter("j_password","weipass");
resp = formname.submit();
System.out.println("Got Page "+resp.getTitle());
link = resp.getLinkWith("Cats");
resp = link.click();
System.out.println("Got Page "+resp.getTitle());
link = resp.getLinkWith("Persian");
resp = link.click();
System.out.println("Got Page "+resp.getTitle());
link = resp.getLinkWith("Add to Cart");
resp = link.click();
System.out.println("Got Page "+resp.getTitle());
link = resp.getLinkWith("Check Out");
resp = link.click();
System.out.println("Got Page "+resp.getTitle());
formname = resp.getFormWithName("order_ino");
resp = formname.submit();
System.out.println("Got Page "+resp.getTitle());
String returnedtitle = resp.getTitle();
assertEquals("Order Placed", returnedtitle);
}
```

TOFT test contains consecutive customer - web site interactions (a customer clicks links and web site returns requested pages). First, HttpUnit uses the returned page title to verify that the expected page is acquired, then it uses getLinkWtih() to select a specific link and clicks the link by calling the click() method. After a series of operations, the customer at last reaches "Order Placed" page, and HttpUnit derives a verdict by comparing the test result against this page title. The parser from TTCN-3 to Java locates the signature in

TTCN-3 which defines the customer - web site interactions, including user login, clicking links and submitting a form with different record types. The parser detects these distinct record type definitions in TTCN-3 and generates the corresponding HttpUnit method calls accordingly. This is how the Java code on page 153 is generated from the TTCN-3 code on page 142.

6.4.4 Boundary

Boundary test contains 5 test methods: test1reg(), test2reg(), test24reg(), test25reg() and test26reg():

```
    public void test1reg() throws Exception{
    String url =
 "http://localhost:8000/petstore/signon_welcome.screen";
    WebLink link;
    WebForm formname;
    WebConversation wc = new WebConversation();
    WebResponse resp = wc.getResponse(url);
    formname = resp.getFormWithName("existingcustomer");
    formname.setParameter("j_username","b");
    formname.setParameter("j_password","password");
    resp = formname.submit();
    System.out.println("Got Page "+resp.getTitle());
    String returnedtitle = resp.getTitle();
    assertEquals("Create Customer", returnedtitle);
    }

    public void test2reg() throws Exception{
    String url =
 "http://localhost:8000/petstore/signon_welcome.screen";
    WebLink link;
    WebForm formname;
    WebConversation wc = new WebConversation();
    WebResponse resp = wc.getResponse(url);
    formname = resp.getFormWithName("existingcustomer");
    formname.setParameter("j_username","b2");
```

```
  formname.setParameter("j_password","password");
  resp = formname.submit();
  System.out.println("Got Page "+resp.getTitle());
  String returnedtitle = resp.getTitle();
  assertEquals("Create Customer", returnedtitle);
  }

  public void test24reg() throws Exception{
  String url =
"http://localhost:8000/petstore/signon_welcome.screen";
  WebLink link;
  WebForm formname;
  WebConversation wc = new WebConversation();
  WebResponse resp = wc.getResponse(url);
  formname = resp.getFormWithName("existingcustomer");
  formname.setParameter("j_username","b111111111111111111111124");
  formname.setParameter("j_password","password");
  resp = formname.submit();
  System.out.println("Got Page "+resp.getTitle());
  String returnedtitle = resp.getTitle();
  assertEquals("Create Customer", returnedtitle);
  }

  public void test25reg() throws Exception{
  String url =
"http://localhost:8000/petstore/signon_welcome.screen";
  WebLink link;
  WebForm formname;
  WebConversation wc = new WebConversation();
  WebResponse resp = wc.getResponse(url);
  formname = resp.getFormWithName("existingcustomer");
  formname.setParameter("j_username","b111111111111111111111125");
  formname.setParameter("j_password","password");
  resp = formname.submit();
  System.out.println("Got Page "+resp.getTitle());
  String returnedtitle = resp.getTitle();
  assertEquals("Create Customer", returnedtitle);
  }
```

```
  public void test26reg() throws Exception{
  String url =
"http://localhost:8000/petstore/signon_welcome.screen";
  WebLink link;
  WebForm formname;
  WebConversation wc = new WebConversation();
  WebResponse resp = wc.getResponse(url);
  formname = resp.getFormWithName("existingcustomer");
  formname.setParameter("j_username","b11111111111111111111111126");
  formname.setParameter("j_password","password");
  resp = formname.submit();
  System.out.println("Got Page "+resp.getTitle());
  String returnedtitle = resp.getTitle();
  assertEquals("Duplicate Account", returnedtitle);
  }
```

Boundary tests focus on the specified test data values. Pet Store requires user name to be 1-25 characters long. Thus, the boundary should be focused on 0, 1, 2, 24, 25, 26 character user names. Here, we deliberately skip test registration with a blank username field for future use of modifying TTCN-3 test scripts in section 6.6, so we choose 1, 2, 24, 25, 26 characters user names to implement Boundary tests. All test cases should return "Create Customer" page except the one using a 26-character user name, which expects a page titled "Duplicate Account".

6.5 Employing Apache Ant to Build, Execute, and Observe Test Results

6.5.1 Apache Ant Build File

We use Apache Ant 1.5 to setup the environment configuration and execute tests. Defined as an XML file, Ant will achieve the following tasks:

- Directory management
- Execute parser to generate Java test code
- Compile test code and execute tests

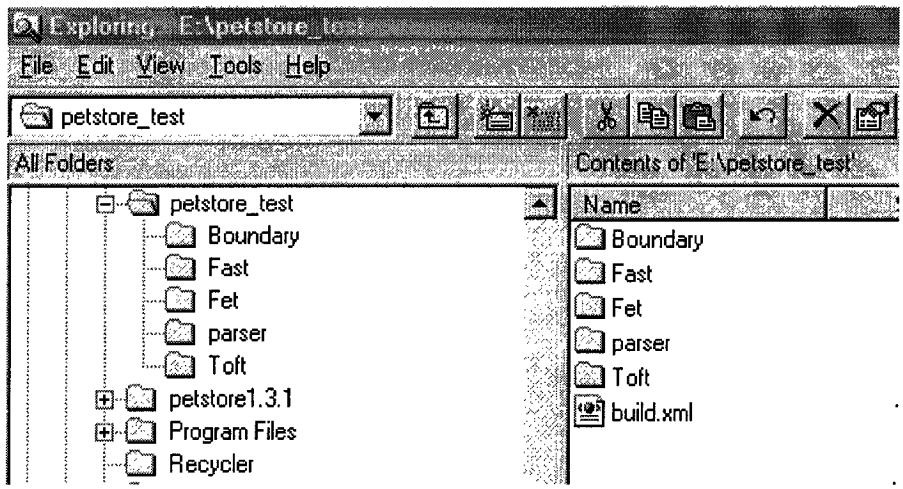

Figure 6.9 The "petstore_test" Directory Before Executing Ant

It is good practice to put all files relevant to testing in a single directory, in our case this is the "petstore_test" folder as shown in Figure 6.9. Different test files of different test design approaches are stored in separate folders. Apache Ant executes the "build.xml" file in the "petstore_test" root directory. For better understanding, we also put our parser classes in this directory. Let's see how the root "build.xml" file works:

```
<?xml version="1.0"?>

<project name="Pet Store functional tests" default="run all" >
    <description>  Doing all functional tests...  </description>

   <target name="run all" >
<ant dir="/petstore_test/FAST" target="run"/>
        <ant dir="/petstore_test/FET" target="run"/>
      <ant dir="/petstore_test/TOFT" target="run"/>
      <ant dir="/petstore_test/Boundary" target="run"/>
  </target>
</project>
```

The <project> tag declares this build file is for Pet Store functional tests and "run all" is the default "target" (target is the functional unit in Ant). Actually this is the only target in this build file. <ant> tags in the "run all" target require the execution of targets of other build files located in the declared directories, in this case, the FAST, FET, TOFT and Boundary test directory. For example, Figure 6.10 shows the directory structure of the Boundary folder where it has its own "build.xml" file.

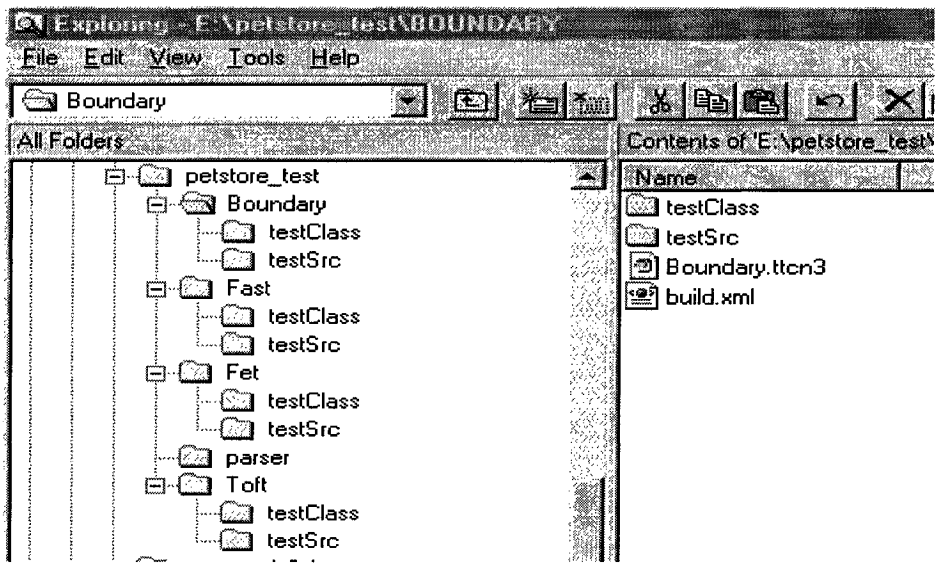

Figure 6.10 The Boundary Directory After Executing Ant

As we have explained in Chapter 3, Boundary tests are designed to test the system variables with boundary values. In our case study, the Boundary tests are configured in the following build file:

```
<?xml version="1.0"?>

<project name="BOUNDARY tests" default="run" >
   <description>   Boundary testing...    </description>

<target name="init">
   <property name="parser" value="/petstore_test/parser" />
   <property name="t3Src" value="/petstore_test/BOUNDARY" />
```

```
     <property name="testSrc" value="/petstore_test/BOUNDARY/testSrc"
/>
     <property name="testClass"
value="/petstore_test/BOUNDARY/testClass"/>
     </target>

  <target name="clean" depends="init">
     <delete dir="${testSrc}" />
     <delete dir="${testClass}" />
  </target>

  <target name="prepare" depends="clean">
     <mkdir dir="${testSrc}" />
     <mkdir dir="${testClass}" />
  </target>

  <target name="gen" depends="prepare">
     <java fork="true" classname="webparser" classpath="${parser}">
          <arg line="${t3Src}/*.ttcn3 ${testSrc}/Boundary.java"/>
     </java>
  </target>

  <target name="compile" depends="gen">
    <javac srcdir="${testSrc}" destdir="${testClass}" />
  </target>

  <target name="run" depends="compile,init">
    <java classname="Boundary" fork="true">
      <classpath>
     <pathelement path="${testClass}"/>
     <pathelement location="e:/junit3.8.1/junit.jar"/>
   <pathelement location="e:\httpunit-1.5.1\lib\httpunit.jar"/>
   <pathelement location="e:\httpunit-1.5.1\jars\js.jar"/>
   <pathelement location="e:\httpunit-1.5.1\jars\junit.jar"/>
   <pathelement location="e:\httpunit-1.5.1\jars\nekohtml.jar"/>
   <pathelement location="e:\httpunit-1.5.1\jars\Tidy.jar"/>
   <pathelement location="e:\httpunit-1.5.1\jars\xercesImpl.jar"/>
   <pathelement location="e:\httpunit-1.5.1\jars\xmlParserAPIs.jar"/>
```

```
        </classpath>
      </java>
    </target>

    </project>
```

The <project> tag defines the scope of our tests and sets default target as "run". In the <init> target, <property> tags define directories for the parser, TTCN-3 test scripts, generated Java test code source files and Java test class files respectively.

The <clean> target deletes existing directories with the specified names and all their contents to get ready for the following targets. The <prepare> target creates "testSrc" and "testClass" directories for Java source files and compiled class files respectively. This will set a tidy and clear environment and avoid different types of files mess up the directory. The <gen> target executes the parser, which parses the TTCN-3 script in the "t3Src" directory. The generated Java test code is saved to previously created "testSrc" directory. At last, Ant compiles the Java test source code, saves the compiled class files in "testClass" directory and executes these tests in the <run> target where the necessary "classpath" are configured.

6.5.2 Case Study Execution

Now we have implemented TTCN-3 test scripts, Ant build files and set up the "petstore_test" directory. We now follow these steps to run the case study:

1. Install J2SKDEE1.3.1, J2SDK1.4.1, Java Pet Store 1.3.1, Apache Ant 1.5 and HttpUnit 1.5.4.
2. Setup the environment including Ant, HttpUnit jar files and parser Java classes in the classpath, setup JAVA_HOME, J2EE_HOME and ANT_HOME system variables
3. Start up Cloudscape [Cloudscape] database server
4. Start up J2EE server [J2EE_RI]
5. Open a DOS prompt, go to "petstore_test" directory and run Ant

6.5.3 Test Observations and Results Reporting

Test Successes

The test root directory build file execution activates executions of build files in sub-directories where test files for different functional test strategies (FAST, FET, TOFT and Boundary) are located. From the test console each "target" along with its "tasks" within this target is shown (see Figure 6.11).

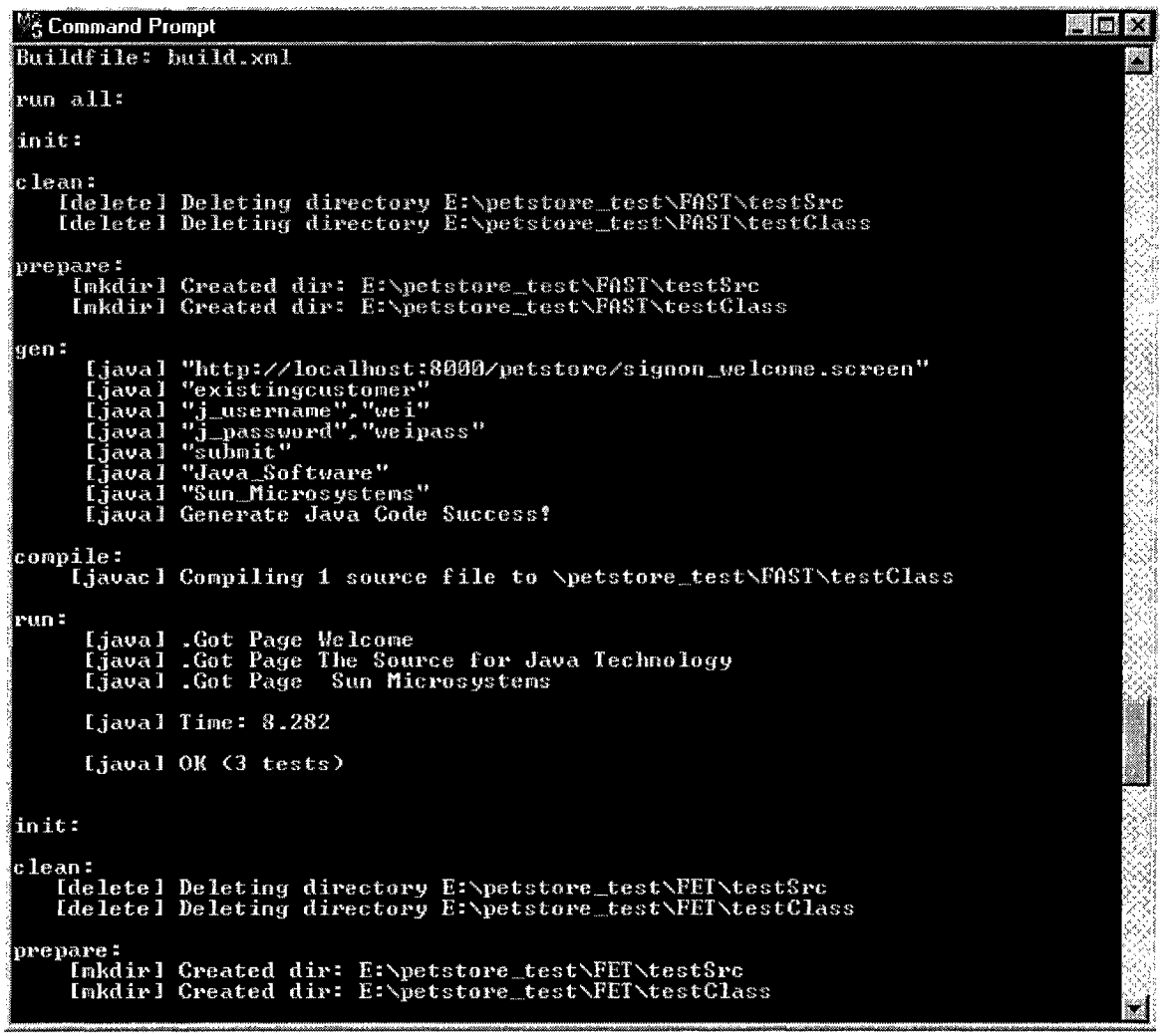

Figure 6.11 Test Success

If we take a closer look at the directory names in Figure 6.11, we can recognize FAST tests as being executed first in the following sequence:

1. the "init" target defines properties
2. the "clean" target deletes existing directories and all their contents (E:\petsore_test\FAST\testSrc and E:\petstore_test\FAST\testClass)
3. the "prepare" target makes new directories "E:\petstore_test\FAST\testSrc" (Java test source code) and "E:\petstore_test\FAST\testClass" (compiled Java class files)
4. the "gen" target parses the TTCN-3 script in "E:\petstore_test\FAST" and saves the parsed Java source code in "E:\petstore_test\FAST\testSrc"
5. the "compile" target compiles the parsed Java code to "E:\petstore_test\testClass"
6. the "run" target executes this Java class file. The expected pages are returned
7. FAST test execution time is 8.282 seconds and this time measurement is logged.
8. the output "[java] OK (3 tests)" indicates all 3 FAST tests are reported as passed

The subsequent targets that stand for FET, TOFT and Boundary tests work in a similar fashion.

Test Failures

If there are test failures, the console displays the failed test case name and the reasons for the failure. Figure 6.12 shows one failure occurred in another FAST test. In the "run" target we get the following output:

```
[java] There was 1 failure:
[java] 1) testcorrectlogin(FAST) junit.framework.ComparisonFailure:
expected:<Welcome> but was:<Sign On Error>
...
[java] FAILURES!!!
[java] Tests run: 3, Failures: 1, Errors: 0
```

The first test case "testcorrectlogin" failed as the expected result is "Welcome" but the actual result is "Sign On Error".

Upon further investigation into the output, we discover in the "gen" target, the value of the "j_username" field is "notwei" instead of "wei" as specified in the test case specification table. This enables us to identify and log the reason for the test case failure.

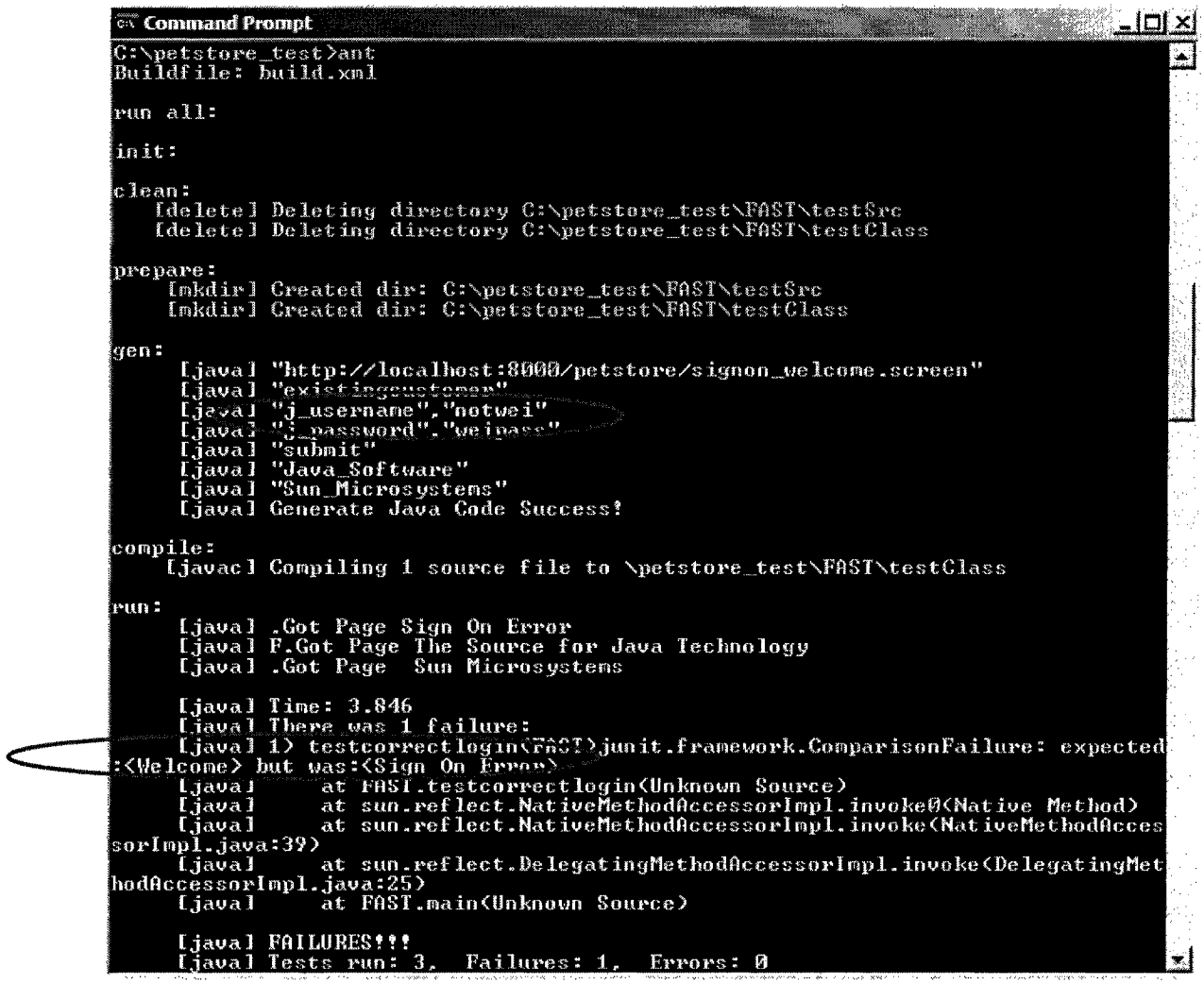

Figure 6.12 Test Failure

6.6 Making Changes in TTCN-3 Script

To demonstrate the flexibility of our extreme programming methodology, we introduce a change in a TTCN-3 test script (which is often the case in the real life software development process) and observe how Apache Ant handles this situation. In the

Boundary tests we chose 5 test cases focused on 1, 2, 24, 25, 26 character registration user names. Figure 6.13 shows the test result. After analyzing the test result, we may realize that a blank user name field registration use case should be tested which we forgot to implement.

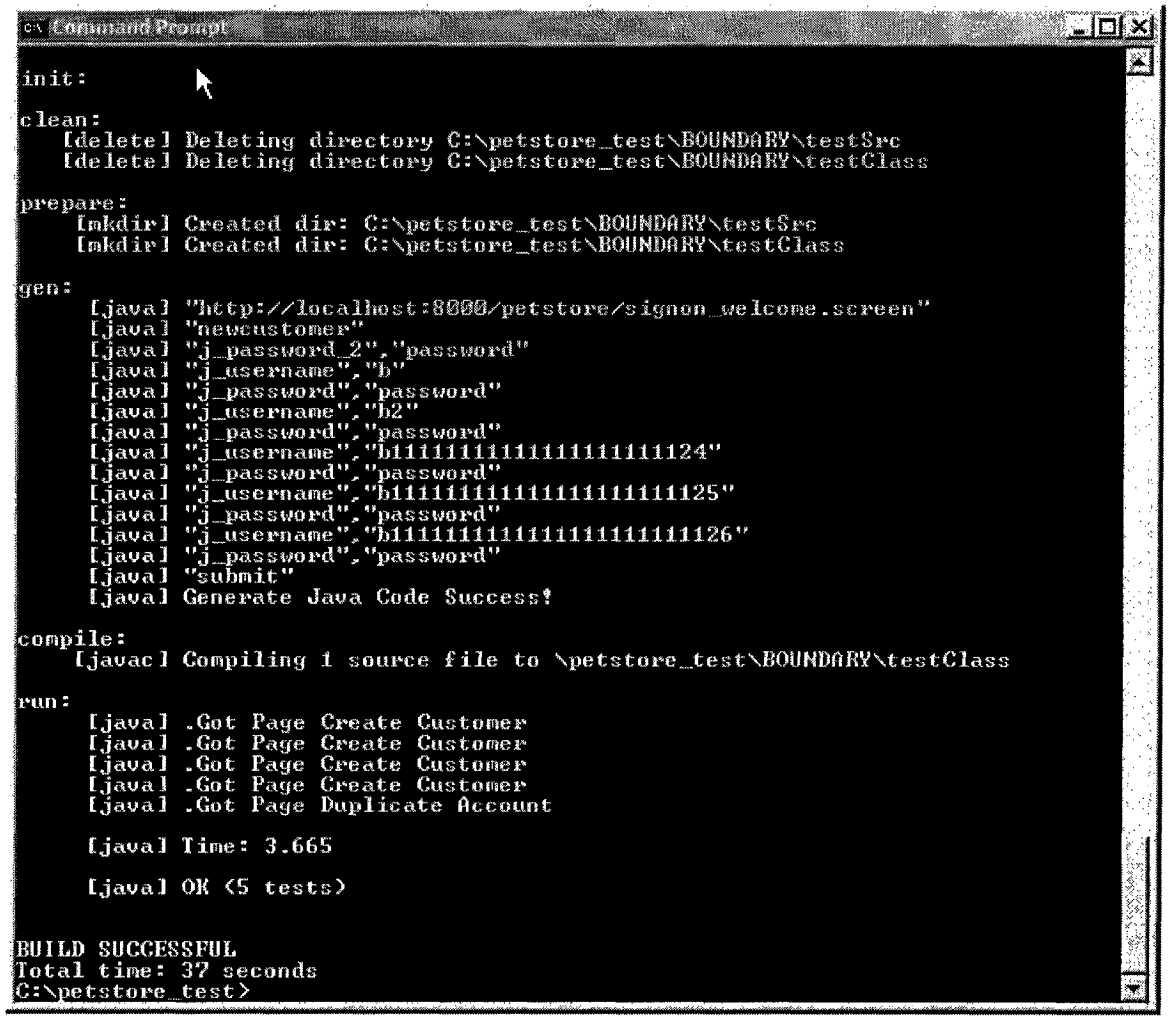

Figure 6.13 Boundary Tests with 5 Test Cases

We add a new test case in Boundary tests in TTCN-3. The test case tests the registration form with a blank user name field. Table 6.13 shows the new test plan entry.

Test Case 6.13 Boundary Test: test0reg

Test case name: test0reg
Test step: see Figure 6.5

Test data:
URL: http://localhost:8000/petstroe/signon_welcome.screen
Form name: newcustomer
User name field name: j_username
User name field value: blank
Password field name: j_password
Password field value: password
Form action: submit
Expected result: return page entitled "Sign On"
Actual result: page entitled "Sign On" returned
Status: Pass

We add the following template and testcase definition into the TTCN-3 Boundary test script:

```
  template myFormType username0:={"j_username", ""};
  template myFormType password0:={"j_password", "password"};


  testcase test0reg runs on MTCType{
  var PTCType formptc;
  formptc.Create;
  connect(formptc:pco, system:tsipco);
  map(formptc:cp, mtc:cp);
  pco.call(testRegCall:{regurl, regformname,  username0,  password0 ,
submit})
  alt{
  []pco.getreply("Sign_On"){
      verdict.set(pass);
      }
    []pco.catch{
      verdict.set(fail);
      }
  stop;
  }
  }
```

In the parsed Java file we derive the following new test method according to the modified TTCN-3 script:

```
  public void test0reg() throws Exception{
  String url =
"http://localhost:8000/petstore/signon_welcome.screen";
  WebLink link;
  WebForm formname;
  WebConversation wc = new WebConversation();
  WebResponse resp = wc.getResponse(url);
  formname = resp.getFormWithName("existingcustomer");
  formname.setParameter("j_username","");
  formname.setParameter("j_password","password");
  resp = formname.submit();
  System.out.println("Got Page "+resp.getTitle());
  String returnedtitle = resp.getTitle();
  assertEquals("Sign On", returnedtitle);
  }
```

The new TTCN-3 test execution result (see Figure 6.14) shows that the additional test case is parsed and executed.

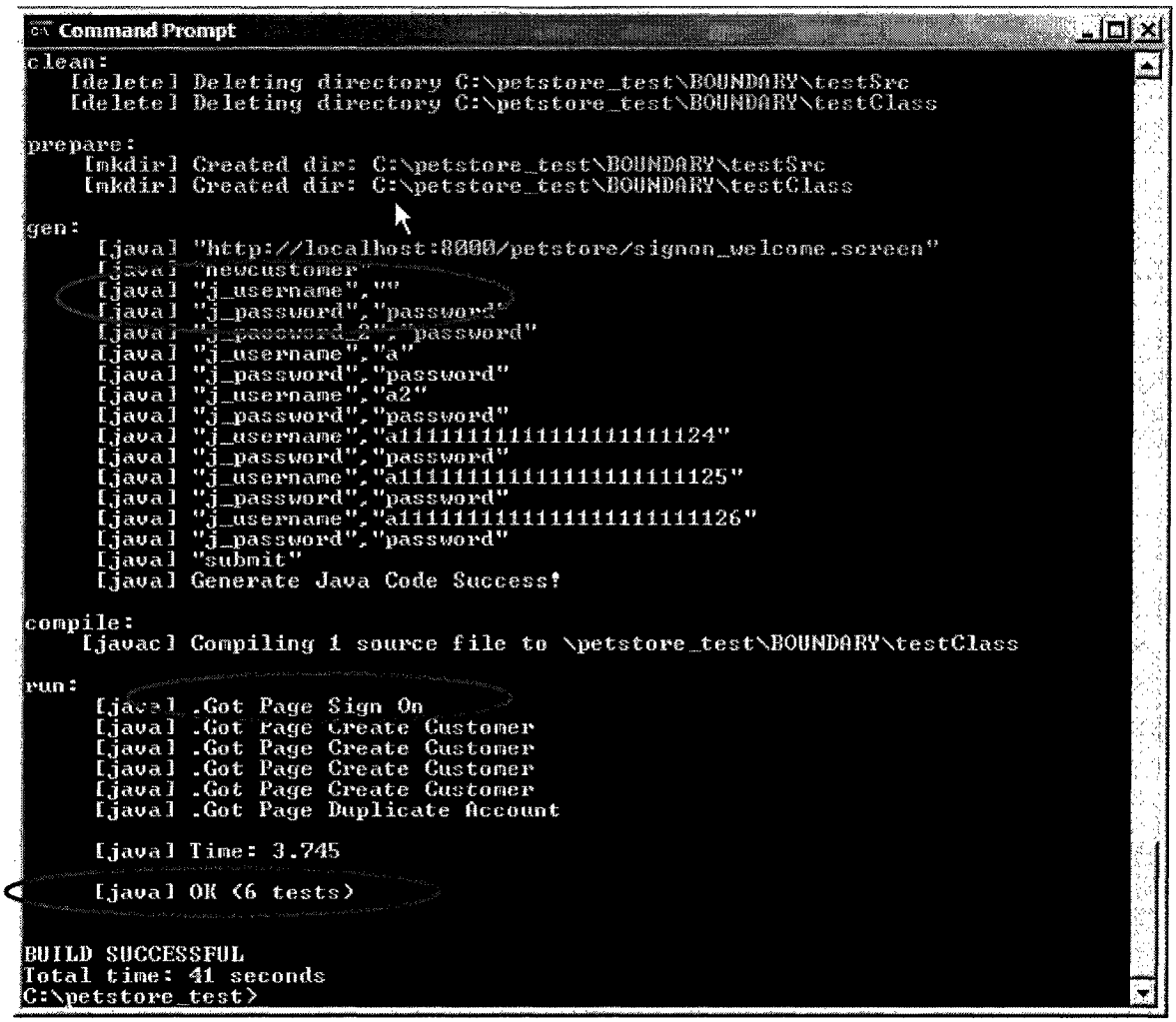

Figure 6.14 Re-run Boundary Tests after Changes Made In TTCN-3 Script

In the "gen" target we can see the output:

```
[java] "j_username", ""
[java] "j_password", "password"
```

The blank user name field indicates new test data is parsed and in the "run" target, we observed the output:

```
[java] .Got Page Sign On.
```

The result "Sign On" indicates the new test case was executed and at the same time, the total Boundary test number was increased to six tests.

6.7 Observations and Results of Case Study

Now we report observations and results of our Case Study by each of the five steps of our approach as shown in Figure 6.15

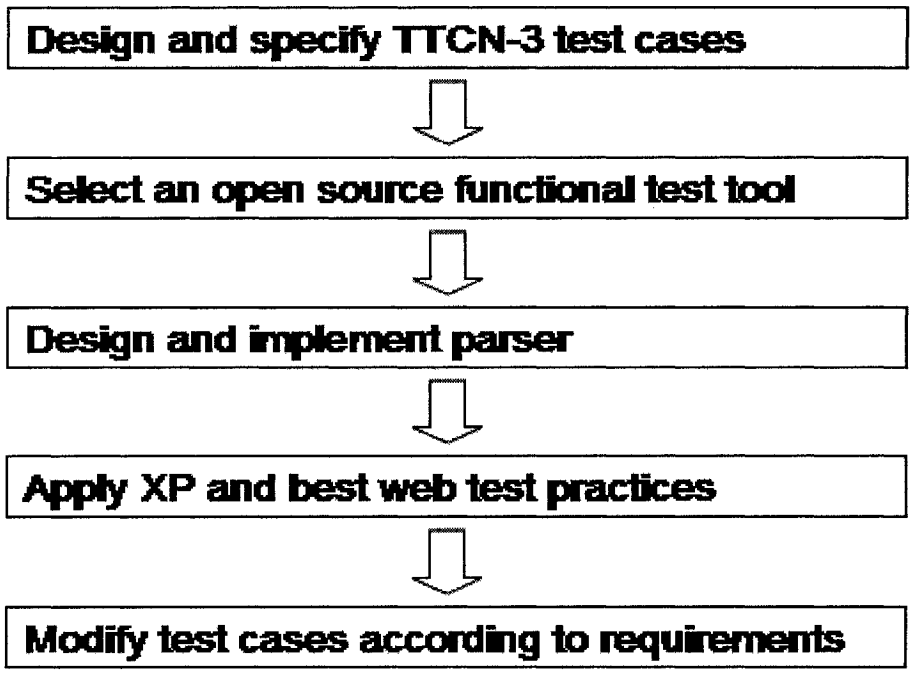

Figure 6.15 Five Steps of Our Approach

Step 1: Design and specify TTCN-3 test cases

To promote the formal test specification language, we used both the graphical and textual formats of TTCN-3 during the test design and specification phase. First, we used MSCs to describe key SUT use cases. The graphical format TTCN-3 gives a clear and natural description of the interactions between the test systems and the SUT and sets up a framework for designing test plans for each test case. Based on previously defined MSCs and detailed test plans, we can implement textual format TTCN-3 test scripts, which use TTCN-3 specific concepts such as "record", "template", "signature", etc. It took one developer 2 weeks to design and write the 13 test cases within this case study.

Step 2: Select an open source functional test tool (HttpUnit)

HttpUnit focuses on end-to-end web-based system functional testing. These tests could be considered as black box testing from the test execution point of view, and yet, they also could be treated as grey-box tests because web page components are exposed to testers who use structural knowledge for test case design. For example, FAST tests in our Case Study have a "customer correct login" test case. Within the "login" page, "login" form name, "username/password" field names and "submit" button names are known. HttpUnit is a set of simple yet powerful APIs that offer the capability to extract web-page information from name/value pairs. Given the exposed web-page component names, it is convenient for us to design and implement HttpUnit test cases to extract corresponding values, especially when we need to embed HttpUnit APIs into customized Java test code.

Step 3: Design and implement parser

The parser in our Case Study aimed at translating textual TTCN-3 scripts to Java test code. The parser extracts test data and test execution constraints defined in TTCN-3 and inserts HttpUnit invocations. This can be seen in sections 6.3 and 6.4. The parsed Java test code reflects the original TTCN-3 test case design and specifications. Although not all TTCN-3 features are used in this Case Study, it shows the feasibility of our approach, namely executing formally specified tests via an intermediate executable language such as Java. The TTCN-3 to Java parser took one developer 4 weeks to design and implement.

Step 4: Apply XP and best web test practices

We applied continuous integration and automatic testing which are two of XP's core practices. Automatic testing is facilitated by HttpUnit, which we described in Step 2. Apache Ant supports continuous integration and one of the advantages we wish to emphasis is test execution efficiency. Short test execution time is important for XP project teams. In our Case Study, the whole test suite (13 test cases) executes within a couple of seconds. This test execution efficiency is obtained by using Apache Ant, which intelligently compiles, deploys and executes tests. Apache Ant can identify if a Java source file is updated by comparing the time stamp of each Java source code with its corresponding binary code, and only new or updated Java files are compiled and

deployed. For example, if we modify a FAST test case in TTCN-3 script and re-ran the test, Apache Ant will only compile the newly parsed Java code, deploy it, and then execute all FAST tests altogether. Although this important test execution efficiency was not so obvious in our Case Study because of the small test suite, it is critical for large test suites containing hundreds of thousands of test cases.

Step 5: Modify test cases according to requirements

Flexible test configuration is another advantage brought about by using Apache Ant. We constructed a "build.xml" file for different test coverage approaches namely, FAST, FET, TOFT and Boundary tests. This XML file defined different tasks that were relevant to the specific test approach. We also wrote a build XML file for executing the whole test suite, which is actually the master build file for all the above sub-layer tests. This test configuration provided us with the flexibility to test the whole test suite or to select and test any sub-layer tests. For example, we can execute the FAST test alone or the FAST and the Boundary test, or execute the whole test suite. In our Case Study, we demonstrated the flexibility of test configuration by adding a new test case in the Boundary test TTCN-3 script and simply running Ant again. Without changing any other part of the test system, we obtained the updated Java test code, the compiled test class and the executed new test suite. From the updated test result we can see the added test case was successfully executed. This flexible test re-configuration will save time and improve test efficiency especially when we need to do minor modifications in a large suite of test cases.

This approach was geared towards separating the work between developers and testers or customers. Testsers or customers can focus on specifying system functional requirements, designing and implementing test cases using both graphical and core TTCN-3, which is an international standard test language. On the other hand, developers can focus on implementing applications using a specific development langugage, such as Java.

6.8 Code Size and Translation Factor

The Case Study contains about 500 lines of TTCN-3 test script, 250 lines of parsed Java test code and 250 lines of Ant XML ("build.xml") code. These web-based functional tests implemented FAST, FET, TOFT and Boundary Interior coverage with a total of 13 test cases (see Table 6.14). Of course our Case Study was not meant to achieve complete test coverage of the SUT, but only as a demonstration of the feasibility of our proposed test approach.

The translation factor is the code size ratio of one language being translated to another. The TTCN-3 to Java translation factor varies depends on different test case design strategies (see Table 6.2)

Table 6.2 Code Size and Translation Factor

Design Strategy	Number of Test cases	Lines of TTCN-3 Script	Lines of Java Code	Java/TTCN-3 Ratio
Boundary	5	186	100	0.54
FAST	3	118	64	0.54
FET	3	127	68	0.54
TOFT	1	80	54	0.66

The Java/TTCN ratio is approximately 0.54 for FAST, TET and Boundary tests. However, for the TOFT test, the ratio is higher. This is because TOFT tests in TTCN-3 describe a sequence of client/server interactions in one TTCN-3 "signature" definition, but the generated Java code will describe these interactions in multiple lines of Java code. For example, in HttpUnit, one HTTP interaction includes an HTTP request and an HTTP response object. The response object needs to call the "getLinkWith()" method to get the "link" object, which further calls the "click()" method to fetch the new response object such as shown in the code segment bellow:

```
link = resp.getLinkWith("Cats");
```

```
resp = link.click();
System.out.println("Got Page "+resp.getTitle());
```

The corresponding TTCN-3 code includes only one line of template definition as bellow:

```
template myLinkType catlink:={"Cats"};
```

It is not uncommon that a TOFT test case contains a number of steps to achieve one single "task" (arriving to the final page). It takes more lines of Java code to describe this "getLink" - "click" action cycle, which means a bigger Java source file is generated by our parser.

Based on our experience, we believe the translation factor will be similar for large test suites (i.e. which means *N* lines of TTCN-3 script will generate *0.54N* lines of Java test code for FAST, FET and Boundary tests. But for TOFT tests, this ratio should be higher (e.g., *0.66N*) depending on the complexity of the specific TOFT test case.

In this chapter we have described the whole process of our Case Study by applying our approach for e-commerce functional testing using TTCN-3. We explained the working mechanism of the system under test, some key scenarios (expressed in MSC notions) that are used in the test process, the TTCN-3 test script and the parsed and generated Java test code. We also demonstrated the test configuration and execution process. Finally, we analyzed test results in terms of successes, failures and feasibility of introducing tests for new requirements into the TTCN-3 test script. The late arrival of new requirements happens in real world development and test practice.

In the next chapter, we analyze our case study data, and conduct detailed comparison of the cost-effectiveness of our approach with an estimated cost-effectiveness of the classical development and test approach.

Chapter 7 Assessment and Conclusions

This chapter assesses our cost-effective functional testing approach using TTCN-3 and pinpoints the advantages and limitations of this effort. In the conclusion part, we summarize this approach in general and set up the scope and condition when applying this approach for e-commerce functional testing. At last, we give some hints and directions for future work.

7.1 Assessment

Our FTEXT approach is a combination of a series of testing activities that promote cost-effective e-commerce functional testing using TTCN-3. We assess our case study in the previous chapter and point out the following advantages and limitations:

7.1.1 Cost-effective Testing with Open Source Tools

This approach is intended for use during development; unfortunately, to reach the widest audience, our case study had to be done using an already developed product. The Sun Pet Store is a well-known sample J2EE application [Sun Pet Store]. Nonetheless, the approach is best applied during development. The cost-effectiveness ratio should actually be even better in real world projects than what we found in our case study.

- Reduced test cost: We introduced Apache Ant and HttpUnit APIs as the main open source tools of our approach. Compared with the counterpart commercial tools [Thurmond00] of similar performance and functionality, open source tools are attractive from a cost perspective to ease the pain of project managers and stake holders who are facing the challenges of limited test tools budget overhead especially under the current market conditions and major budget reductions. Also, with the increasing acceptance level of open source tools, commercial software vendors and managers will be expected to show value and ROI for their products [Weiss02]. The ultimate beneficiary would be the user of such test tools, namely the design engineer or developer.

- Simple and flexible to use: With the growth of the Web and the error-prone nature of Web programming, many commercial test tool products hit the market with elaborate GUIs to guide the developer through the testing process [Hower]. The open-source HttpUnit on the other hand, is free of licensing fees and very simple to use. HttpUnit maintains client state, sending requests, retrieving responses and following links. Also, it provides handy shortcut methods to extract meaningful elements from an HTTP response, like headers, tables etc. In addition, the availability of the HttpUnit Java source code and its simple APIs let developers create their own testing solutions to fit the needs of their particular organizations. Using HttpUnit APIs as a foundation, we can build complex test suites at minimal cost.

- Free from vender lock in [Kucharik03]: Commercial test products often have their own sets of APIs. Customers of such tools become tied to the proprietary interfaces of a single vendor. If the vendor raises prices or fails to fix bugs in their tools, the customer cannot easily switch to another product without rewriting their test code. Our approach avoided being locked to HttpUnit and even TTCN-3. Designing and implementing a new parser is a cost-effective and flexible approach to bridge a specific test languge and test tool.

- Standardized/de-facto tool: An open source tools like Apache Ant is becoming the standardized or de-facto build tool for Java application. It is integrated into both commercial and open source popular Java IDEs like Borland JBuilder [Borland], NetBeans [NetBeans] and Eclipse [Eclipse]. What is more, Apache Ant offers functionalities that no commercial counterparts can provide.

- Cost effective quality: savings will accrue by the use of open source tools. Open source advocates believe that the open source model encourages several activities that are not common in the development of commercial tools. For example, open source developers not only report bugs but also actually track down their root

causes and fix them. The source code is open to everybody, a great many developers review each other's code, and this peer review is the most effective way to find defects [Connel03]. Another factor that improves open source software quality is that open source projects don't face the same type of resource and time pressures that commercial projects do. Open source projects are rarely developed against a fixed timeline, affording more opportunity for peer review, and usually offer extensive beta testing before releases.

7.1.2 Bridging TTCN-3 and Java

Functional tests are different from unit tests in that they test the system from end-to-end and most importantly, the customer is involved in creating the functional tests. Ideally, a framework should be built that allows the customer the full ability to add or drop tests for the system according to their test goals. Customers who implement functional tests may not share the same language with developers. In our case, customers may use TTCN-3 to address functional tests. Bridging the gap between Java and TTCN-3 makes the testing architecture highly scaleable to accommodate different test languages.

7.1.3 Using International Standard Formal Test Specification Language

In addition to the merits of TTCN-3 itself, the combination of HttpUnit and TTCN-3 brings about other advantages, for example, a typical test case for a Web page that contains a form might be to fill out the form in several wrong ways, verifying that validation works in each case, and then to fill it out correctly, checking that the appropriate subsequent page is displayed. But writing such Java code to perform these tests could become cumbersome, because much of it would be repeated with minor changes (which form fields to fill out, which failure message was returned, and so on). This shortcoming could be overcome by using TTCN-3. With the "template" structure, we can simply change the proper template without modifying other parts of the TTCN-3 test script file, and then the TTCN-3 to Java parser will automatically generate new Java

test code. Also, these newly generated test cases are automatically compiled, deployed and executed with the help of the ANT build tool.

7.1.4 Applying Web Functional Test Best Practices

We followed a set of web functional test best practices namely FAST, TOFT, FET and Boundary testing in this approach. In the real world, there exists many more "best practices" with respect to different points of view from our testing practices.

7.1.5 Limitations and Scope

We identify the following limitations and scope of this approach:

- Our parser is tool dependent. The HttpUnit API is based on the JUnit test framework and focused on Web functional tests. If we need to do performance testing using JUnitPerf or JMeter, a new parser will have to be implemented in order to translate TTCN-3 to Java code that applies to these open source tools. But with the development and maturity of open source tools, we could argue, the selection of the best breed tool will be focused on a limited number of tools that facilitate on a certain task, like Apache Ant which is becoming the de facto Java build tool in the Java community.
- Our focus is on functional test: The case study focuses on functional testing as we have a complete working system under test in hand. For a typical extreme programming project, during the project design and implementation stage, unit test should be the main concern and should be carried out throughout the development process. Furthermore, performance and stress/load testing is probably just as critical especially for web-based e-commerce systems.

7.2 Conclusions and Future Work

7.2.1 Conclusions

This thesis promotes and facilitates the use of TTCN-3 for e-commerce system functional testing. By applying extreme programming methodology, developing a parser that translates TTCN-3 script to Java test code and by following a set of best practice

functional test strategies we have developed what appears to be a cost-effective approach to facilitate e-commerce functional testing using TTCN-3. To demonstrate our test approach, we implemented a Case Study that targets a typical Java-based e-commerce system as the system under test. The Case Study achieved the following test objectives:

- Demonstrated the use of the international standard test language -TTCN-3 for e-commerce functional testing.
- Showed how to adopt the extreme programming software development methodology, which often seem to cope better with software development reality. The most valuable core practices are automatic testing and continuous integration, which embrace changes during the development process.
- Showed how to design and develop a parser that translates TTCN-3 script to Java test code appropriate to a selected open source test tool, HttpUnit.
- Showed how to select and apply open source test tools. These tools are free and often of higher quality as they are developed and maintained by various developers. Also, some of these open source tools are standardized in the Java community.
- Presented and showed how to apply Web functional test best practice strategies. These test strategies cover functional tests from in different perspectives [Nguyen03].
- Analyzed the Case Study and concluded that this test approach meets the expected objectives and it also opens a new window for applying TTCN-3 to e-commerce test applications.

7.2.2 Future Work

As we mentioned in the above thesis conclusion, the main objective of this approach is to promote TTCN-3, formal testing, and the XP methodology for e-commerce testing. We identify the following areas that need future work:

Apply other open source test tools

We explained in the "Big picture" section in chapter 5 that different open source tools applied to different stages of the J2EE e-commerce system development process. The case study in this thesis focuses on functional test of the overall system using HttpUnit

APIs, yet, at the basic building block class implementation phase, JUnit could be applied to unit test basic Java classes; and Cactus could be used to test functional behavior in the J2EE container environment. We may implement parsers that translate TTCN-3 test scripts to Java test files applicable for these open source test tools.

Improve test automation and integration

At present, test scripts are manually generated in TTCN-3. Other options include automatically generating TTCN-3 from UML or SDL during the requirement specification stage and the design stage (Telelogic Tau [Telelogic Tau] has an older version that automatically generate TTCN-2 test suites). Also, there are still improvements required in test result report and analysis such as integrated tools that generate graphically test results and automatically distribute them to project development team members or customers.

Facilitate performance and load testing

We focused on web functional testing in our case study, another critical and sometimes more important aspect of an e-commerce system may be the performance testing. JUnitPerf and JMeter could be good candidates for performance tests.

Avoid re-inventing the wheel

Best practices either in development or testing are the results of many real world project experiences and practices. They are of extreme high value in improving the quality and efficiency of software development. The discovery and exploration of such best practices is an ongoing process along the history and practice of software development. Design Patterns [Gamma95] and frameworks are development examples of these best practices. There are "patterns and frameworks" for testing as well. Following such best practices proven practices will benefit testers.

Apply formal method specification and tools

TTCN-3 is part of the formal method approach, which includes a set of methodologies and tools along the software development process. For example, in the requirement

analysis phase, UML and MSC are widely accepted as a means to specify system requirements and customer-system interactions. In the development phase, SDL is a powerful tool used to develop test logic and test processes. We may implement interfaces between these test languages and tools that automatically translate UML and MSC in the front end and SDL in the backend into TTCN-3 test suites. We then apply our test approach as a subsequent test process. This enhances the test approach by facilitating the use of formal methods for specification. The use of open source test tools will also improve the overall software quality in a cost-effective manner.

Apply to other enterprise architectures

All the objectives in the short-term perspective aimed at systems built upon J2EE architecture. We may customize this approach to other middle-ware architectures like CORBA, Microsoft .NET and especially Web Services as they will also play an important role in future e-commerce system developments.

REFERENCES

[Ant] Apache Ant official web site http://ant.apache.org/

[Beck01] Kent Beck, Martin Fowler, *Planning Extreme programming.* Addison-Wesley, 2001

[Beck98] Kent Beck, Erich Gamma, *Test Infected: Programmers Love Writing Tests*, Java Report, July 1998, volume 3, number 7, pp. 37–50.

[Beck99] Kent Beck, *Extreme Programming Explained: Embrace Change.* Addison-Wesley 1999.

[Blexrud00] Christopher Blexrud et al, *Professional Windows DNA: Building Distributed Web Applications with VB, COM+, MSMQ, SOAP, and ASP*, Wrox Press 1st edition, September 2000.

[Blueprints] Sun Website, http://java.sun.com/blueprints/enterprise/.

[Boehm81] B. W. Boehm. *Software Engineering Economics*, Prentice-Hall, 1981.

[Borland] Borland Website, http://www.borland.com

[Cactus] Cactus Website, http://jakarta.apache.org/cactus/how_it_works.html

[Clarke96] E. M. Clarke and J. M. Wing. *Formal Methods: State of the Art and Future Directions*, ACM Computing Surveys, December 1996.

[Cloudscape] IBM Website, http://www-306.ibm.com/software/data/cloudscape/

[Con98] ISO/IEC 9646-3 (1998): "Information technology - Open systems interconnection - Conformance testing methodology and framework - Part 3: *The Tree and Tabular combined Notation* (TTCN)"

[Connell03] Charles Connell, *All Source Code Should Be Open.* *http://www.developer.com/open/article.php/1452091*

[Cowen99] R. Cowen, *NASA loses Mars Climate Orbiter.* Science News 156(Oct. 2): 214. 1999.

[Eclipse] Eclipse Website, http://www.eclipse.org

[Empirix] Empirix eLoad Web site http://www.empirix.com/Empirix

[ETSI01] ETSI MTS: The Testing and Test Control Notation TTCN-3, Part 1: TTCN-3 Core Language / ETSI ES 201873-1V2.0.0 (2001-03), http://www.etsi.org

[evalid] eValid Inc. Website, http://www.soft.com/eValid

[Fowler00] Martin Fowler, *The New Methodology*, September 2000. http://www.martinfowler.com/articles/newMethodology.html

[Fowler99] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison Wesley Longman, 1999.

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns.* Addison-Wesley, 1995

[Grabowski03] J. Grabowski, D. Hogrefe, G. Rethy, I. Schieferdecker, A. Wiles, C. Willcock, *An Introduction into the Testing and Test Control Notation (TTCN-3)*, Computer Networks Journal, Vol. 42, Issue 3, 2003.

[Hower] Rick Hower, *Web Site Test Tools and Site Management Tools,* http://www.softwareqatest.com/qatweb1.html

[HttpUnit] HttpUnit web site http://www.httpunit.org/

[IEEE90] Institute of Electrical and Electronics Engineers, *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, New York, 1990.

[Infoworld03] Open Source Code Quality Endorsed, Infoworld, July 2003.

[J2EE_RI] J2EE reference implementation Website, http://java.sun.com/j2ee/

[J2EE_spe] Sun Microsystems, Java™ 2 Platform Enterprise Edition Specification, v1.4, 2003

[Jacobson99] Ivar Jacobson, Grady Booch, James Rumbaugh, *The Unified Software Development Process*, Addison Wesley, 1999.

[JCC] Code Conventions for the Java Programming Language, http://java.sun.com/docs/codeconv/.

[Jeffries00] Ron Jeffries, Ann Anderson, Chet Hendrickson, *Extreme Programming Installed.* Addison-Wesley 1st edition, 2000.

[Johnson03] Rod Johnson, *Expert one-on-one J2EE design and development.* Wiley Publishing, Inc. 2003

[JUnit] JUnit official Website, http://www.junit.org.

[JUnitPerf] JUnitPerf official Website, www.clarkware.com/software/JUnitPerf.html

[Kerievsky02] Joshua Kerievsky. *The Extreme Programming Playbook.* Industrial Logic, Inc. 2002

[Kruchten00] Philippe Kruchten. *The Rational Unified Process: An Introduction (2nd Edition),* Addison-Wesley, March 2000.

[Kucharik03] Amy Kucharik, *Proprietary and lock-in not necessarily synonymous,* SearchEnterpriseLinux.com, July 2003.

[Mercury] LoadRunner Website, www.mercury.com

[Microsoft] MicroSoft Website, http://www.microsoft.com/net/

[Myers79] Myers, G. J. *The Art of Software Testing.* John Wiley & Sons Inc, 1979.

[NetBeans] NetBeans Website, http://www.netbeans.org

[Nguyen03] Hung Q. Nguyen, Bob Johnson, Michael Hackett. *Testing applications on the web, Second edition.* Wiley Publishing Inc, 2003.

[OMG] Object Management Group Website, www.omg.org

[Ostrand88] Ostrand, T, and BALCER, M, *The category-partition method for specifying and generating functional tests,* Communications of the ACM, 1988.

[PushToTest] PushToTest TestMaker Website, http://www.pushtotest.com

[RadView] RadView Software Ltd. Website, http://www.radview.com

[Roper94] Roper, M, *Software Testing.* McGraw Hill Book Company, 1994.

[Schieferdecker01a] I.Schieferdecker, S.Pietsch, T.Vassiliou-Gioles, *Systematic Testing of Internet Protocols – First Experiences in using TTCN-3 for SIP.* Proc. of the 5th Africom Conf. on Communication Systems, Cape Town, South Africa, May 2001.

[Schieferdecker01b] Ina Schieferdecker, *GMD Report 153*, 2001

[Schieferdecker03] I. Schieferdecker, B. Stepien, *Automated Testing of XML/SOAP based Web Services,* 13. Fachkonferenz der Gesellschaft fuer Informatik (GI) Fachgruppe "Kommunikation in verteilten Systemen" (KiVS), Leipzig, 26.-28. Febr. 2003.

[Schulz02] S. Schulz, T. Vassiliou-Gioles, *Implementation of TTCN-3 Test Systems using the TRI.* IFIP 14th International Conference on Testing of Communicating Systems (TestCom 2002), Berlin (Germany), March 2002.

[Segue] SilkPerformer Website, www.segue.com/html/s_solutions/ s_performer/s_performer.htm

[Semantic] Semantic Designs, Inc. Website, http://www.semdesigns.com/Products/Formatters/index.html

[SOAP1.1] The World Wide Web Consortium, http://www.w3.org/tr/soap/

[Sommerville92] Sommerville, I., *Software engineering*, Addison-Wesley Publishing Company Inc., 1992

[Succi02] Succi, G, Marchesi, M., Pedrycz,W., Williams, L., *Preliminary Analysis of the Effects of Pair Programming on Job Satisfaction*, Fourth International Conference on eXtreme Programming and Agile Processes in Software Engineering, 2002.

[Sun Pet Store] Sun Website, http://java.sun.com/blueprints/code/jps131/docs/

[Telelogic] Telelogic Tau Website, www.telelogic.com

[Thurmond00] David Thurmond, *Surveying the Open Source Landscape.* 2000

[Turban03] Efraim Turban, David King, Jae K. Lee, Dennis Viehland, *Electronic Commerce: A Managerial Perspective*, Prentice Hall, 2003.

[Udo01] Godwin J. Udo, *Privacy and security concern as major barriers for e-commerce: a survey study, Information Management & Computer Security*, MCB University Press, 2001.

[Weiss02] Todd R.Weiss, *Vendors to Push ROI at Linux World.* Computer World, August 2002

[Wiles02] A. Wiles, T. Vassiliou-Gioles, S. Moseley, S. Mueller, *Experiences of Using TTCN-3 for Testing SIP and OSP*, 1st ATS Conference, ACATS Forum Conference, Athens, March 2002.

ACRONYMS

B-ISDN	Broadband-Integrated Services Digital Network
ATM	Asynchronous Transfer Mode
AAT	Agenda Authoring Tool
ASN.1	Abstract System Notation one
ABT	Application Being Tested
ADO	ActiveX Data Object
API	Application Programming Interface
B2B	Business To Business
B2C	Business To Customer
CGI	Common Gateway Interface
CIL	Common Intermediate Language
CLR	Common Language Runtime
COM	Component Object Model
CTMF	Conformance Testing Methodology and Framework
CORBA	Component Object Request Broker Architecture
DCOM	Distributed Common Object Model
DII	Dynamic Invocation Interface
DNA	Distributed interNet Applications Architecture
DTD	Data Type Definition
DTSTTCPW	Do The Simplest Thing That Could Possibly Work
EAR	Enterprise Archive Repository
EDI	Electronic Data Interchange
EJB	Enterprise Java Bean
ERP	Enterprise Resource Plan
ETSI	European Telecommunication Standard Institute

FAST	Functional Acceptance Simple Test
FDT	Formal Description Technique
FET	Forced Error Test
FET	Forced Error Test
FIFO	First In First Out
FSM	Finite State Machines
FTP	File Transfer Protocol
GSM	Global System for Mobile Communications
HTTP	Hyper Text Transfer Protocol
IDE	Integrated Development Environment
IDL	Interface Description Language
IEEE	Institute of Electrical and Electronics Engineers
IIOP	Inter Internet ORB Protocol
ISO	International Organization for Standardization
SIP	Session Initiation Protocol
IPv6	Internet Protocol Version 6
ITU	International Telecommunication Union
J2EE	Java 2 Enterprise Edition
JAR	Java Archive Repository
JDBC	Java Database Connectivity
JCP	Java Community Process
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JRE	Java Runtime Environnent
JSP	Java Server Page
MTC	Main Test Component
MSC	Message Sequence Chart
MSMQ	Microsoft Message Queue
MTS	Microsoft Transaction Server
NAT	Network Address Translation
ORB	Objecte Request Broker

PTC	Parallel Test Component
QA	Quality Assurance
RAD	Rapid Application Development
RMI	Remote Method Innovation
ROI	Return Of Investment
RPC	Remote Procedure Call
RUP	Rational Unified Process
SMTP	Simple Mail Transfer Protocol
SDL	Specification and Description Language
SUT	System Under Test
TSC	Test Sequence Chart
CCITT	International Consultative Committee for Telephony and Telegraphy
TOFT	Task Oriented Functional Test
TTCN	Tree and Tabular Combined Notation
TTCN-3	Test and Test Control Notation version 3
UDDI	Universal Description, Discovery, and Integration
UML	Unified Modeling Language
URL	Universal Resource Location
WAR	Web Archive Repository
WSDL	Web Services Description Language
XML	eXtended Markup Language
XP	eXtreme Programming
YAGNI	You Aren't Going To Need It

Appendix A: TTCN-3 Test Script

```
/*
Name: TOFT.ttcn3
Version 1.0
Author: Wei Xu
Last update: 09/20/2003
Notes: This test module contains 1 testcase-testplaceorder
*/

module TOFT{

type port CPType message{
inout ReportType;
}
type port PCOtype procedure{
inout testLoginCall, testBrowseCall, testOrderCall;
}

type component MTCType{
  port CPType cp;
}

type component PTCType{
 port PCOType pco;
 port CPType cp;
}

type component TestSystemType{
 port PCOType tsi_pco
}

type record ReportType{
 integer passnr;
 integer failnr;
}
```

```
   function ResultAcc() return ReportType{
   cp.receive(verdict);
   if (verdict:=pass){
      passnr:=passnr+1;
   }else{
      failnr:=failnr+1;
   };
   return;
   }

   type record myUrlType{charstring url};
   type record myFormNameType{charstring formname};
   type record myFormType{charstring field};
   type record myLinkType{charstring link};
   type record mySubmitType{charstring submit}

   template myUrlType
loginurl:={"http://localhost:8000/petstore/signon_welcome.screen"};
   template myLinkType signinlink:={"Sign_in"};
   template myFormNameType loginformname:={"existingcustomer"};
   template myFormType correctusername:={"j_username", "wei"};
   template myFormType correctpassword:={"j_password", "weipass"};
   template mySubmitType submit:={"submit"};
   template myLinkType catlink:={"Cats"};
   template myLinkType persianlink:={"Persian"};
   template myLinkType addtocartlink:={"Add_to_Cart"};
   template myLinkType checkoutlink:={"Check_Out"};
   template myFormNameType submitformname:={"order_ino"};

   signature testOrderCall(myUrlType url, myLinkType link0,
myFormnameType,fName, mySubmitType sub, myLinkType link1, myLinkType
link2, myLinkType link3, myLinkType link4);

   testcase testplaceorder runs on MTCType{
   var PTCType placeorderptc;
   placeorderptc.Create;
   connect(placeorderptc:pco, system:tsipco);
```

```
  map(placeorderptc:cp, mtc:cp);
  pco.call(testOrderCall:{loginurl,signinlink, loginformname,
correctusername, correctpassword, submit, catlink, persianlink,
addtocartlink, checkoutlink, submitformname, submit})
  alt{
    []pco.getreply("Order_Placed"){
      verdict.set(pass);
      }
    []pco.catch{
      verdict.set(fail);
      }
  stop;
  }
  }

  control{
  execute(testorder());
  ResultAcc();
  }
  }

  /*
  Name: FET.ttcn3
  Version 1.0
  Author: Wei Xu
  Last update: 09/20/2003
  Notes: This test module contains 3 testcases, testblankusername,
  testblandpassword and testwroinglogin. All should return a "Sign on
  error" page
  */

  module FET{

  type port CPType message{
  inout ReportType;
  }
  type port PCOtype procedure{
  inout testLoginCall, testBrowseCall, testOrderCall;
```

```
}

type component MTCType{
  port CPType cp;
}

type component PTCType{
 port PCOType pco;
 port CPType cp;
}

type component TestSystemType{
 port PCOType tsi_pco
}

type record ReportType{
 integer passnr;
 integer failnr;
}

function ResultAcc() return ReportType{
cp.receive(verdict);
if (verdict:=pass){
   passnr:=passnr+1;
}else{
   failnr:=failnr+1;
};
return;
}

type record myUrlType{charstring url};
type record myFormNameType{charstring formname};
type record myFormType{charstring field};
type record myLinkType{charstring link};
type record mySubmitType{charstring submit}

template myUrlType
loginurl:={"http://localhost:8000/petstore/signon_welcome.screen"};
```

```
template myFormNameType loginformname:={"existingcustomer"};

template myFormType blankusername:={"j_username", ""};
template myFormType password:={"j_password", "password"};

template myFormType username:={"j_username", "wei"};
template myFormType blankpassword:={"j_password", ""};

template myFormType wrongusername:={"j_username", "notwei"};
template myFormType wrongpassword:={"j_password", "notweipass"};

template mySubmitType submit:={"submit"};

signature testLoginCall{myUrlType url, myFormNameType fName,
myFormType uName,myFormType pwd, mySubmitType sub};

testcase testblankusernamelogin runs on MTCType{
var PTCType formptc;
formptc.Create;
connect(formptc:pco, system:tsipco);
map(formptc:cp, mtc:cp);
pco.call(testLoginCall:{loginurl,loginformname, blankusername,
password,submit})
alt{
[]pco.getreply("Sign_On"){
    verdict.set(pass);
    }
  []pco.catch{
    verdict.set(fail);
    }
stop;
}
}

testcase testblankpasswordlogin runs on MTCType{
var PTCType formptc;
formptc.Create;
connect(formptc:pco, system:tsipco);
```

```
map(formptc:cp, mtc:cp);
 pco.call(testLoginCall:{loginurl,loginformname,  username,
 blankpassword,submit})
alt{
[]pco.getreply("Sign_On"){
    verdict.set(pass);
    }
  []pco.catch{
    verdict.set(fail);
    }
stop;
}
}

testcase testwronglogin runs on{
var PTCType formptc;
formptc.Create;
connect(formptc:pco, system:tsipco);
map(formptc:cp, mtc:cp);
 pco.call(testLoginCall:{loginurl, loginformname, wrongusername,
 wrongpassword, submit})
alt{
  []pco.getreply("Sign_On_Error"){
    verdict.set(pass);
    }
  []pco.catch{
    verdict.set(fail);
    }
stop;
}
}

control{
var ReportType report:={passnr:=0, failnr:=0};
execute(testblankusernamelogin());
ResultAcc();
execute(testblandpasswordlogin());
ResultAcc();
```

```
execute(testwronglogin());
ResultAcc();
}
}

/*
Name: Boundary.ttcn3
Version 1.0
Author: Wei Xu
Last update: 09/20/2003
Notes: This test module is to test user name field with 1, 2, 24,
25, 26 characters in the login process. Users login with 1, 2,24,25
character user names should return a "Welcome" page. Users login
with 26 characters should return a "Sign on error" page.
*/

module Boundary{

type port CPType message{
inout ReportType;
}
type port PCOtype procedure{
inout testLoginCall, testBrowseCall, testOrderCall;
}

type component MTCType{
  port CPType cp;
}

type component PTCType{
 port PCOType pco;
 port CPType cp;
}

type component TestSystemType{
 port PCOType tsi_pco
}
```

```
type record ReportType{
 integer passnr;
 integer failnr;
}

function ResultAcc() return ReportType{
cp.receive(verdict);
if (verdict:=pass){
   passnr:=passnr+1;
}else{
   failnr:=failnr+1;
};
return;
}

type record myUrlType{charstring url};
type record myFormNameType{charstring formname};
type record myFormType{charstring field};
type record myLinkType{charstring link};
type record mySubmitType{charstring submit}

template myUrlType
loginurl:={"http://localhost:8000/petstore/signon_welcome.screen"};
template myFormNameType loginformname:={"existingcustomer"};
template myFormType  username1:={"j_username", "1"};
template myFormType  password1:={"j_password", "password"};

template myFormType  username2:={"j_username", "22"};
template myFormType  password2:={"j_password", "password"};

template myFormType  username24:={"j_username",
"111111111111111111111124"};
template myFormType  password24:={"j_password", "password"};

template myFormType  username25:={"j_username",
"1111111111111111111111125"};
template myFormType  password25:={"j_password", "password"};
```

```
template myFormType  username26:={"j_username",
"11111111111111111111111126"};
template myFormType  password26:={"j_password", "password"};

template mySubmitType submit:={"submit"};

signature testLoginCall{myUrlType url, myFormNameType fName,
myFormType uName,myFormType pwd, mySubmitType sub};

testcase test1login runs on MTCType{
var PTCType formptc;
formptc.Create;
connect(formptc:pco, system:tsipco);
map(formptc:cp, mtc:cp);
pco.call(testLoginCall:{loginurl, loginformname,  username1,
password1, submit})
alt{
[]pco.getreply("Welcome"){
    verdict.set(pass);
    }
  []pco.catch{
    verdict.set(fail);
    }
stop;
}
}

testcase test2login runs on MTCType{
var PTCType formptc;
formptc.Create;
connect(formptc:pco, system:tsipco);
map(formptc:cp, mtc:cp);
pco.call(testLoginCall:{loginurl,loginformname, username2,
password2,submit})
alt{
[]pco.getreply("Welcome"){
    verdict.set(pass);
    }
```

```
  []pco.catch{
    verdict.set(fail);
    }
stop;
}
}

testcase test24login runs on MTCType{
var PTCType formptc;
formptc.Create;
connect(formptc:pco, system:tsipco);
map(formptc:cp, mtc:cp);
pco.call(testLoginCall:{loginurl,loginformname, username24,
password24,submit})
alt{
[]pco.getreply("Welcome"){
    verdict.set(pass);
    }
  []pco.catch{
    verdict.set(fail);
    }
stop;
}
}

testcase test25login runs on MTCType{
var PTCType formptc;
formptc.Create;
connect(formptc:pco, system:tsipco);
map(formptc:cp, mtc:cp);
pco.call(testLoginCall:{loginurl,loginformname, username25,
password25,submit})
alt{
[]pco.getreply("Welcome"){
    verdict.set(pass);
    }
  []pco.catch{
    verdict.set(fail);
```

```
    }
stop;
}
}

testcase test26login runs on{
var PTCType formptc;
formptc.Create;
connect(formptc:pco, system:tsipco);
map(formptc:cp, mtc:cp);
pco.call(testLoginCall:{loginurl, loginformname, username26,
password26, submit})
alt{
  []pco.getreply("Sign_On_Error"){
    verdict.set(pass);
    }
  []pco.catch{
    verdict.set(fail);
    }
stop;
}
}
control{
var ReportType report:={passnr:=0, failnr:=0};
execute(test1login());
ResultAcc();
execute(test2login());
ResultAcc();
execute(test24login());
ResultAcc();
execute(test25login());
ResultAcc();
execute(test26login());
ResultAcc();
}
}
```

Appendix B: Generated Java code

```
/*
Name: FAST.java
Version 1.0
Author: Wei Xu
Last update: 09/20/2003
Notes: this is the parsed Java code, which is used in httpUnit test
*/

import com.meterware.httpunit.*;
import junit.framework.*;
import java.util.*;

public class FAST extends TestCase {

public FAST(java.lang.String testName) {
super(testName);
}

public static Test suite() {
TestSuite suite = new TestSuite(FAST.class);
return suite;
}

public static void main(java.lang.String[] args) {
junit.textui.TestRunner.run(suite());
}

public void testcorrectlogin() throws Exception{
String url =
"http://localhost:8000/petstore/signon_welcome.screen";
WebLink link;
WebForm formname;
WebConversation wc = new WebConversation();
```

```
WebResponse resp = wc.getResponse(url);
formname = resp.getFormWithName("existingcustomer");
formname.setParameter("j_username","wei");
formname.setParameter("j_password","weipass");
resp = formname.submit();
System.out.println("Got Page "+resp.getTitle());
String returnedtitle = resp.getTitle();
assertEquals("Welcome", returnedtitle);
}

public void testjavasoftlink() throws Exception{
String url =
"http://localhost:8000/petstore/signon_welcome.screen";
WebLink link;
WebForm formname;
WebConversation wc = new WebConversation();
WebResponse resp = wc.getResponse(url);
link = resp.getLinkWith("Java Software");
resp = link.click();
System.out.println("Got Page "+resp.getTitle());
String returnedtitle = resp.getTitle();
assertEquals("The Source for Java Technology", returnedtitle);
}

public void testsunlink() throws Exception{
String url =
"http://localhost:8000/petstore/signon_welcome.screen";
WebLink link;
WebForm formname;
WebConversation wc = new WebConversation();
WebResponse resp = wc.getResponse(url);
link = resp.getLinkWith("Sun Microsystems");
resp = link.click();
System.out.println("Got Page "+resp.getTitle());
String returnedtitle = resp.getTitle();
assertEquals(" Sun Microsystems ", returnedtitle);
}
}
```

```
/*
Name: TOFT.java
Version 1.0
Author: Wei Xu
Last update: 09/20/2003
Notes: this is the parsed Java code, which is used in httpUnit test
*/

import com.meterware.httpunit.*;
import junit.framework.*;
import java.util.*;

public class TOFT extends TestCase {

public TOFT(java.lang.String testName) {
super(testName);
}

public static Test suite() {
TestSuite suite = new TestSuite(TOFT.class);
return suite;
}

public static void main(java.lang.String[] args) {
junit.textui.TestRunner.run(suite());
}

public void testplaceorder() throws Exception{
String url =
"http://localhost:8000/petstore/signon_welcome.screen";
WebLink link;
WebForm formname;
WebConversation wc = new WebConversation();
WebResponse resp = wc.getResponse(url);
link = resp.getLinkWith("Sign in");
resp = link.click();
```

```
System.out.println("Got Page "+resp.getTitle());
formname = resp.getFormWithName("existingcustomer");
formname.setParameter("j_username","wei");
formname.setParameter("j_password","weipass");
resp = formname.submit();
System.out.println("Got Page "+resp.getTitle());
link = resp.getLinkWith("Cats");
resp = link.click();
System.out.println("Got Page "+resp.getTitle());
link = resp.getLinkWith("Persian");
resp = link.click();
System.out.println("Got Page "+resp.getTitle());
link = resp.getLinkWith("Add to Cart");
resp = link.click();
System.out.println("Got Page "+resp.getTitle());
link = resp.getLinkWith("Check Out");
resp = link.click();
System.out.println("Got Page "+resp.getTitle());
formname = resp.getFormWithName("order_ino");
resp = formname.submit();
System.out.println("Got Page "+resp.getTitle());
String returnedtitle = resp.getTitle();
assertEquals("Order Placed", returnedtitle);
}

}

/*
Name: FET.java
Version 1.0
Author: Wei Xu
Last update: 09/20/2003
Notes: this is the parsed Java code, which is used in httpUnit test
*/

import com.meterware.httpunit.*;
import junit.framework.*;
import java.util.*;
```

```
public class FET extends TestCase {

public FET(java.lang.String testName) {
super(testName);
}

public static Test suite() {
TestSuite suite = new TestSuite(FET.class);
return suite;
}

public static void main(java.lang.String[] args) {
junit.textui.TestRunner.run(suite());
}

public void testblankusernamelogin() throws Exception{
String url =
"http://localhost:8000/petstore/signon_welcome.screen";
WebLink link;
WebForm formname;
WebConversation wc = new WebConversation();
WebResponse resp = wc.getResponse(url);
formname = resp.getFormWithName("existingcustomer");
formname.setParameter("j_username","");
formname.setParameter("j_password","password");
resp = formname.submit();
System.out.println("Got Page "+resp.getTitle());
String returnedtitle = resp.getTitle();
assertEquals("Sign On", returnedtitle);
}

public void testblankpasswordlogin() throws Exception{
String url =
"http://localhost:8000/petstore/signon_welcome.screen";
WebLink link;
WebForm formname;
WebConversation wc = new WebConversation();
```

```
WebResponse resp = wc.getResponse(url);
formname = resp.getFormWithName("existingcustomer");
formname.setParameter("j_username","wei");
formname.setParameter("j_password","");
resp = formname.submit();
System.out.println("Got Page "+resp.getTitle());
String returnedtitle = resp.getTitle();
assertEquals("Sign On", returnedtitle);
}

public void testwronglogin() throws Exception{
String url =
"http://localhost:8000/petstore/signon_welcome.screen";
WebLink link;
WebForm formname;
WebConversation wc = new WebConversation();
WebResponse resp = wc.getResponse(url);
formname = resp.getFormWithName("existingcustomer");
formname.setParameter("j_username","notwei");
formname.setParameter("j_password","notweipass");
resp = formname.submit();
System.out.println("Got Page "+resp.getTitle());
String returnedtitle = resp.getTitle();
assertEquals("Sign On Error", returnedtitle);
}
}

/*
Name: Boundary.java
Version 1.0
Author: Wei Xu
Last update: 09/20/2003
Notes: this is the parsed Java code, which is used in httpUnit test
*/

import com.meterware.httpunit.*;
import junit.framework.*;
import java.util.*;
```

```
public class Boundary extends TestCase {

public Boundary(java.lang.String testName) {
super(testName);
}

public static Test suite() {
TestSuite suite = new TestSuite(Boundary.class);
return suite;
}

public static void main(java.lang.String[] args) {
junit.textui.TestRunner.run(suite());
}

public void test1login() throws Exception{
String url =
"http://localhost:8000/petstore/signon_welcome.screen";
WebLink link;
WebForm formname;
WebConversation wc = new WebConversation();
WebResponse resp = wc.getResponse(url);
formname = resp.getFormWithName("existingcustomer");
formname.setParameter("j_username","1");
formname.setParameter("j_password","password");
resp = formname.submit();
System.out.println("Got Page "+resp.getTitle());
String returnedtitle = resp.getTitle();
assertEquals("Welcome", returnedtitle);
}

public void test2login() throws Exception{
String url =
"http://localhost:8000/petstore/signon_welcome.screen";
WebLink link;
WebForm formname;
WebConversation wc = new WebConversation();
```

```
WebResponse resp = wc.getResponse(url);
formname = resp.getFormWithName("existingcustomer");
formname.setParameter("j_username","22");
formname.setParameter("j_password","password");
resp = formname.submit();
System.out.println("Got Page "+resp.getTitle());
String returnedtitle = resp.getTitle();
assertEquals("Welcome", returnedtitle);
}

public void test24login() throws Exception{
String url =
"http://localhost:8000/petstore/signon_welcome.screen";
WebLink link;
WebForm formname;
WebConversation wc = new WebConversation();
WebResponse resp = wc.getResponse(url);
formname = resp.getFormWithName("existingcustomer");
formname.setParameter("j_username","111111111111111111111124");
formname.setParameter("j_password","password");
resp = formname.submit();
System.out.println("Got Page "+resp.getTitle());
String returnedtitle = resp.getTitle();
assertEquals("Welcome", returnedtitle);
}

public void test25login() throws Exception{
String url =
"http://localhost:8000/petstore/signon_welcome.screen";
WebLink link;
WebForm formname;
WebConversation wc = new WebConversation();
WebResponse resp = wc.getResponse(url);
formname = resp.getFormWithName("existingcustomer");
formname.setParameter("j_username","111111111111111111111125");
formname.setParameter("j_password","password");
resp = formname.submit();
System.out.println("Got Page "+resp.getTitle());
```

```
String returnedtitle = resp.getTitle();
assertEquals("Welcome", returnedtitle);
}

public void test26login() throws Exception{
String url =
"http://localhost:8000/petstore/signon_welcome.screen";
WebLink link;
WebForm formname;
WebConversation wc = new WebConversation();
WebResponse resp = wc.getResponse(url);
formname = resp.getFormWithName("existingcustomer");
formname.setParameter("j_username","11111111111111111111111126");
formname.setParameter("j_password","password");
resp = formname.submit();
System.out.println("Got Page "+resp.getTitle());
String returnedtitle = resp.getTitle();
assertEquals("Sign On Error", returnedtitle);
}
}
```

Appendix C: Ant Build File

```
/*
Name: build.xml
Version 1.0
Author: Wei Xu
Last update: 09/20/2003
Notes: this is the build file for all tests.
*/

<?xml version="1.0"?>

<project name="Pet Store functinal test" default="run all" >
    <description>        Doing all fucntional tests...
</description>

<target name="run all" >
    <ant dir="/petstore_test/FET" target="run"/>
    <ant dir="/petstore_test/TOFT" target="run"/>
    <ant dir="/petstore_test/Boundary" target="run"/>
</target>
</project>


/*
Name: build.xml
Version 1.0
Author: Wei Xu
Last update: 09/20/2003
Notes: this is the build file for Boundary tests.
*/

<?xml version="1.0"?>

<project name="BOUNDARY tests" default="run" >
   <description>       Boundary testing...    </description>
```

```
<target name="init">
      <property name="parser" value="/petstore_test/parser" />
      <property name="t3Src" value="/petstore_test/BOUNDARY" />
      <property name="testSrc"
value="/petstore_test/BOUNDARY/testSrc" />
      <property name="testClass"
value="/petstore_test/BOUNDARY/testClass" />
</target>

<target name="clean" depends="init">
      <delete dir="${testSrc}" />
      <delete dir="${testClass}" />
</target>

<target name="prepare" depends="clean">
      <mkdir dir="${testSrc}" />
      <mkdir dir="${testClass}" />
</target>

<target name="gen" depends="prepare">
         <java  fork="true" classname="webparser"
classpath="${parser}">
        <!--arg line="${t3Src}/*.ttcn3 *.java"/-->
<arg line="${t3Src}/*.ttcn3 ${testSrc}/Boundary.java"/>
         </java>
</target>

<target name="compile" depends="gen">
   <javac srcdir="${testSrc}" destdir="${testClass}" />
</target>

<target name="run" depends="compile,init">
    <java classname="Boundary" fork="true">
        <classpath>
        <pathelement path="${testClass}"/>
        <pathelement location="e:/junit3.8.1/junit.jar"/>
<pathelement location="e:\httpunit-1.5.1\lib\httpunit.jar"/>
```

```
<pathelement location="e:\httpunit-1.5.1\jars\js.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\junit.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\nekohtml.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\Tidy.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\xercesImpl.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\xmlParserAPIs.jar"/>
       </classpath>
      </java>
</target>
</project>


/*
Name: build.xml
Version 1.0
Author: Wei Xu
Last update: 09/20/2003
Notes: this is the build file for FET tests.
*/

<?xml version="1.0"?>

<project name="FET tests" default="run" >
  <description>       FET testing...    </description>

<target name="init">
      <property name="parser" value="/petstore_test/parser" />
      <property name="t3Src" value="/petstore_test/FET" />
      <property name="testSrc" value="/petstore_test/FET/testSrc"
/>
      <property name="testClass"
value="/petstore_test/FET/testClass" />
</target>

<target name="clean" depends="init">
      <delete dir="${testSrc}" />
      <delete dir="${testClass}" />
```

```
</target>

<target name="prepare" depends="clean">
      <mkdir dir="${testSrc}" />
      <mkdir dir="${testClass}" />
</target>

<target name="gen" depends="prepare">
         <java  fork="true" classname="webparser"
classpath="${parser}">
        <!--arg line="${t3Src}/*.ttcn3 *.java"/-->
<arg line="${t3Src}/*.ttcn3 ${testSrc}/FET.java"/>
         </java>
</target>

<target name="compile" depends="gen">
      <javac srcdir="${testSrc}" destdir="${testClass}" />
</target>

<target name="run" depends="compile,init">
        <java classname="FET" fork="true">
              <classpath>
        <pathelement path="${testClass}"/>
        <pathelement location="e:/junit3.8.1/junit.jar"/>
<pathelement location="e:\httpunit-1.5.1\lib\httpunit.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\js.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\junit.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\nekohtml.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\Tidy.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\xercesImpl.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\xmlParserAPIs.jar"/>
              </classpath>
        </java>
</target>
</project>


/*
```

```
Name: build.xml
Version 1.0
Author: Wei Xu
Last update: 09/20/2003
Notes: this is the build file for TOFT tests.
*/

<?xml version="1.0"?>
<project name="TOFT tests" default="run" >
   <description>       TOFT testing...    </description>

<target name="init">
      <property name="parser" value="/petstore_test/parser" />
      <property name="t3Src" value="/petstore_test/TOFT" />
      <property name="testSrc" value="/petstore_test/TOFT/testSrc"
/>
      <property name="testClass"
value="/petstore_test/TOFT/testClass" />
</target>

<target name="clean" depends="init">
      <delete dir="${testSrc}" />
      <delete dir="${testClass}" />

</target>

<target name="prepare" depends="clean">
      <mkdir dir="${testSrc}" />
      <mkdir dir="${testClass}" />
</target>

<target name="gen" depends="prepare">
         <java  fork="true" classname="webparser"
classpath="${parser}">
        <!--arg line="${t3Src}/*.ttcn3 *.java"/-->
<arg line="${t3Src}/*.ttcn3 ${testSrc}/TOFT.java"/>
         </java>
</target>
```

```
<target name="compile" depends="gen">
     <javac srcdir="${testSrc}" destdir="${testClass}" />
</target>

<target name="run" depends="compile,init">
         <java classname="TOFT" fork="true">
           <classpath>
        <pathelement path="${testClass}"/>
        <pathelement location="e:/junit3.8.1/junit.jar"/>
<pathelement location="e:\httpunit-1.5.1\lib\httpunit.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\js.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\junit.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\nekohtml.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\Tidy.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\xercesImpl.jar"/>
<pathelement location="e:\httpunit-1.5.1\jars\xmlParserAPIs.jar"/>
           </classpath>
         </java>
</target>
</project>
```

Appendix D: Parser Source Code

```
/*
Name: Webparser.java
Version 1.0
Author: Wei Xu
Last update: 09/20/2003
Notes: Combined with helper class Handler.java, this program parses
a TTCN-3 file to Java code, which is used in httpUnit test.
*/

import java.io.*;
import java.util.*;

public class Webparser{

public static void main(String[] args) throws IOException {

if (args[0]==null)
System.out.println("Please name the ttcn3 file.");
else if (args[1]==null)  System.out.println("Please name the java
file.");

       File inputFile = new File(args[0]);
       File outputFile = new File(args[1]);

       FileReader in = new FileReader(inputFile);
       FileWriter out = new FileWriter(outputFile);
       int c;
StringBuffer sb=new StringBuffer();
      while ((c = in.read()) != -1)
sb.append((char)c);
String str=sb.toString();

StringTokenizer stn=new StringTokenizer(str, "
\t\n\r\f(){};:=.,_\"", true);
```

```
String[] word1=new String[stn.countTokens()];

for(int i=0;i<word1.length;i++) {
word1[i]=(String)stn.nextToken();
}

ArrayList al=new ArrayList();

for(int i=0;i<word1.length;i++){
  if(!(word1[i].equals(" ")))
   al.add(word1[i]);
}

String[] word=new String[al.size()];
for(int k=0;k<word.length;k++) {
word[k]=(String)al.get(k);
}

Handler handler = new Handler();

//start processing keywords
for(int i=0;i<word.length;i++){

if(word[i].equals("type")){
  int firstf = findfirst(i, "{", word);
  int endf = fef(firstf, word);
  i = endf;
  }

if(word[i].equals("//")){
  int firstnewline = findfirst(i, "\n", word);
  i=firstnewline;
  }// end of processing c++ style comments.

if(word[i].equals("/*")){
  int firstend =  findfirst(i, "*/", word);
  i = firstend;
```

```
  }
// end of processing c style comments.

if(word[i].equals("signature")){
  int firstnewline = findfirst(i, "\n", word);
  i=firstnewline;
  }

if(word[i].equals("module")){
handler.setaclassname(word[i+1]);
}

if(word[i].equals("template")){
HashMap hm = new HashMap();
ArrayList alist = new ArrayList();
String temptype = word[i+1];
String tempname = word[i+2];
int fieldstartpos = findfirst(i, "{", word);
int fieldendpos = fef(fieldstartpos, word);
String tempvalue = content(fieldstartpos, fieldendpos, word);
System.out.println(tempvalue+" ");
alist.add(temptype);
alist.add(tempvalue);
hm.put(tempname, alist);
handler.settempal(hm);
}
//end of processing template

if(word[i].equals("testcase")){
String testcasename = word[i+1];
handler.settestcasename(testcasename);

HashMap testcasehm = new HashMap();
ArrayList callal = new ArrayList();

int reppos = findfirst(i, "getreply", word);
int repfirstp = findfirst(reppos, "(", word);
int rependp = fep(repfirstp, word);
```

```
String rep = content(repfirstp, rependp, word);
String newrep = rep.replaceAll("_", " ");
handler.setareply(newrep);

//find and save methodcall parameters to callal
int callpos = findfirst(i, "call", word);
String signame = word[callpos+2];
int flowerstartpos = findfirst(callpos, "{", word);
int flowerendpos = fef(flowerstartpos, word);
int j=flowerstartpos+1;
while(j<=flowerendpos-2){
  int cpos = findfirst(j, ",", word);
  String s = content(j-1, cpos, word);
  j=cpos+1;
  callal.add(s);
}
String last = word[flowerendpos-1];
callal.add(last);
testcasehm.put(testcasename, callal);
handler.settestcase(testcasehm);
}//end of processing testcase
}//end extracting data to Arrays

//start using saved data to construct output

//construct imports
out.write("import com.meterware.httpunit.*;\n");
out.write("import junit.framework.*;\n");
out.write("import java.util.*;\n\n");

//construct class name
String className = handler.getaclassname();
out.write("public class "+className+" extends TestCase {\n\n");

//construct constructor
out.write("public "+className+"(java.lang.String testName) {\n");
out.write("super(testName);\n");
out.write("}\n\n");
```

```
//construct suite method
 out.write("public static Test suite() {\n");
 out.write("TestSuite suite = new
 TestSuite("+className+".class);\n");
 out.write("return suite;\n");
 out.write("}\n\n");

//construct main method
out.write("public static void main(java.lang.String[] args) {\n");
out.write("junit.textui.TestRunner.run(suite());\n");
out.write("}\n\n");

//construct testcases
for(int k=0; k<handler.testcasenumber(); k++){
String testcasename = handler.gettestcasename(k);
out.write("public void "+testcasename+"() throws Exception{\n");

HashMap hm = new HashMap();
hm = (HashMap)handler.atestcase.get(k);
ArrayList lastelement = new ArrayList();
lastelement = (ArrayList)hm.get(testcasename);

for(int m=0; m<lastelement.size(); m++){
  for(int n=0; n<handler.tempal.size(); n++){
    String lastv = lastelement.get(m).toString();
    if(handler.gettempal(n).containsKey(lastv)){
  ArrayList myal = (ArrayList)handler.gettempal(n).get(lastv);

String key = myal.get(0).toString();
String value = myal.get(1).toString();
 if(key.equals("myUrlType")) {
  String url = value;
        out.write("String url = "+url+";\n");
  out.write("WebLink link;\n");
  out.write("WebForm formname;\n");
  out.write("WebConversation wc = new WebConversation();\n");
  out.write("WebResponse resp = wc.getResponse(url);\n");
```

```
 }

 if(key.equals("myFormNameType")) {
  String formname = value;
  out.write("formname = resp.getFormWithName("+formname+");\n");
 }

 if(key.equals("myFormType")) {
  String form = value;
  out.write("formname.setParameter("+form+");\n");
 }

 if(key.equals("mySubmitType")) {
  String submit = value;
  out.write("resp = formname.submit();\n");
  out.write("System.out.println(\"Got Page
  \"+resp.getTitle());\n");
  }


 if(key.equals("myLinkType")) {
  String oldlink = value;
  String link = oldlink.replaceAll("_", " ");
  out.write("link = resp.getLinkWith("+link+");\n");
  out.write("resp = link.click();\n");
  out.write("System.out.println(\"Got Page
  \"+resp.getTitle());\n");
 }

 }//end if containsKey

 }//end for tempal

}//end for lastelemental=testcaseal

out.write("String returnedtitle = resp.getTitle();\n");
//out.write("String newreturnedtitle = returnedtitle.replaceAll(\"
\", \"\");\n");
```

```
    out.write("assertEquals("+handler.areply.get(k)+",
    returnedtitle);\n");
    out.write("}\n\n");

   }//end construct testcases

   out.write("}");
   in.close();
   out.close();

   }//end main


    /*findfirst(int start, String note, String[] word) returns the
    first specified character following the "start" position
    */

   public static int findfirst(int start, String note, String[] word){
          int pos=0;
          for(int i=start;i<word.length;i++){
     if(word[i].equals(note)) { pos=i; break;}
          }
     if(pos==0) {System.out.println("cant find ***"+note+"***
"+start); return Integer.MAX_VALUE;}
     else return pos;
   }

   /*fef(int from, String[] wrod) returns the counterpart end "}"
   position following the "from" position.
   */
   public static int fef(int from, String[] word){
      int counter=0, pos=0;
      for(int i=from; i<word.length; i++){
      if(word[i].equals("{")) counter++;
      if(word[i].equals("}")) counter--;
      if(counter==0) {pos=i; break;}
     }
```

```
    if(pos==0) {System.out.println("cant find counterpart \"}\"");
  return -1;}
  else return pos;
  }

  /*
  fep(int from, String[] word) returns the counterpart parenthesis
  position //following the "from" position.
  */

  public static int fep(int from, String[] word){
     int counter=0, pos=0;
     for(int i=from; i<word.length; i++){
     if(word[i].equals("(")) counter++;
     if(word[i].equals(")")) counter--;
     if(counter==0) {pos=i; break;}
    }
     if(pos==0) {System.out.println("cant find counterpart \")\"");
  return -1;}
  else return pos;
  }

/*
  content(int start, int end, String[] word) fetches the content string
  between start and end position
*/

    public static String content(int start, int end, String[] word){
     StringBuffer buf=new StringBuffer();
     for(int i=start+1;i<end;i++){
     buf.append(word[i]);
    }
     return buf.toString();
    }
    }
    //end of class webparser
```

```
/*
Name: Handler.java
Version 1.0
Author: Wei Xu
Last update: 09/20/2003
Notes: This is the helper class that parses a TTCN-3 file to Java
code, which is used in httpUnit test.
*/

import java.util.*;
import java.io.*;
import java.util.*;

public class Handler {

ArrayList aclassname = new ArrayList();
public void setaclassname(String s){
aclassname.add(s);
}

public String getaclassname(){
return aclassname.get(0).toString();
}

ArrayList areply = new ArrayList();
public void setareply (String s){
areply.add(s);
}

public String getareply(int i){
return areply.get(i).toString();
}

ArrayList atestcasename = new ArrayList();
public void settestcasename(String testcasename){
atestcasename.add(testcasename);
}
```

```
    public String gettestcasename(int i){
    return atestcasename.get(i).toString();
    }

    public int testcasenumber(){
    return atestcasename.size();
    }

    ArrayList atestcase = new ArrayList();
    public void settestcase(HashMap hm){
    atestcase.add(hm);
    }

    ArrayList tempal = new ArrayList();
    public void settempal(HashMap hm){
    tempal.add(hm);
    }

    public HashMap gettempal(int i){
    return (HashMap)tempal.get(i);
    }
}// end of class Handler
```