"The new software testing methods introduced in this book will show you how to form and apply effective online goal-directed design and testing techniques. You begin with your user's goals, follow Frank's recommendations for scalable system design, and end with powerful tests that measure your user's success in achieving them."

—Alan Cooper,
Author of *About Face 2.0: The Essentials of Interaction Design*
and *The Inmates Are Running the Asylum.*
He is also the father of Visual Basic

"A must read for Java developers needing to deliver high-quality, scalable, and well performing production Web Services."

—Phil Goodwin,
Staff Engineer, JAX Team, Sun Microsystems, Inc.

"Testing, scalability, and test-driven design and methodologies are the new 'battlefield' of software development. Frank Cohen introduces a new methodology for designing, testing, and maintaining Web-enabled applications. This book is the best battlegear available."

—Andrew Sliwkowski,
Performance Analyst, BEA Systems

"Frank uses his considerable experience in information system technology to provide a view of the evolution from mainframe to Client/Server to distributed 'n' or 'flapjack' tier architectures, structured to object-oriented design, and waterfall to Agile/Scrum/XP development approaches. Now that we (as an industry) have moved toward this distributed, open environment—testing, scalability, and interoperability are the next challenges, and Frank's book is an excellent introduction."

—Ed Hunter,
SunOne Performance Team, Sun Microsystems, Inc.

"The book is an excellent presentation of metrics and measurement approaches to harnessing and calculating performance and functionality when working with Web-enabled applications, especially Web Services. The book's emphasis on user archetypes to drive otherwise headless testing agents is a key point."

—Bret Pettichord,
Software Testing and Test Automation Guru for IBM/Tivoli, BMC, Segue,
Interleaf; and co-author of *Lessons Learned in Software Testing*

"Frank Cohen's new book takes a theoretical, didactic, and empirical approach to attacking this issue and does a fantastic job at bringing both the novice and expert from the 'what and why' to the 'how' of software testing design, automation, and scalability/reusability."

—Christian J. Hessler
Staff Engineer, Sun Microsystems, Inc.

"An unequaled achievement in clarity of Java testing, design, and optimization. To avoid hard-to-trace scalability and performance 'gotchas' inherent in early Java development, this book is for you!"

—Farooq Khan,
Software Development Manager, 2Wire Inc.

"What I liked most about this book was the confidence with which Cohen presents a wide range of topics: objects, tests, architecture, frameworks, and design—all the right stuff for a software developer, QA technician, and IT manager."

—Rossana Muriel,
Director of Quality, Advise & Services, AMP Global Assets Management

Java™ Testing and Design

From Unit Testing to Automated Web Tests

Frank Cohen

PRENTICE HALL PTR

PRENTICE HALL
PROFESSIONAL TECHNICAL REFERENCE
UPPER SADDLE RIVER, NJ 07458
WWW.PHPTR.COM

To Lorette—truly, deeply, madly

❖ ❖ ❖

A CIP catalog reference for this book can be obtained from the Library of Congress

Editorial/Production Supervision: *MetroVoice Publishing Services*
Executive Editor: *Greg Doench*
Marketing Manager: *Chris Guzikowski*
Manufacturing Manager: *Alexis Heydt-Long*
Cover Designer: *Nina Scuderi*
Cover Design Director: *Jerry Votta*
Series Designer: *Gail Cocker-Bogusz*
Full-Service Project Manager: *Anne R. Garcia*

PRENTICE HALL PTR

© 2004 by Pearson Education, Inc.
Publishing as Prentice Hall Professional Technical Reference
Upper Saddle River, New Jersey 07458

Printed in the United States of America

First Printing

ISBN 0-13-142189-1

Pearson Education LTD.
Pearson Education Australia PTY, Limited
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd.
Pearson Education Canada, Ltd.
Pearson Educación de Mexico, S.A. de C.V.
Pearson Education—Japan
Pearson Education Malaysia, Pte. Ltd.

Contents

Foreword **xiii**

Preface **xv**

Acknowledgments **xix**

PART I
Gauging Web-Enabled Applications

CHAPTER 1
The Forces at Work Affecting Your Web-Enabled Software **1**

The First Three Axioms 2
Web-Enabled Applications, the New Frontier 4
Why Writing High-Quality Software Today Is Hard 10
The Myth of Version 2.0 Solidity 11
Management's Quest for Grail 12
Trying for Homogeneity When Heterogeneity Rules 15
The Language of Bugs 17
The Evil Twin Vice Presidents Problem 18
A Concise History of Software Development 19
Web-Enabled Applications 20
Test Paradigms and Lifecycle Processes 22
Testing Methods 25
Click-Stream Testing 26
Unit Testing 27
Functional System Testing 28
Scalability and Performance Testing 30
Quality of Service Testing 30

Defining Test Agents 30
Scalability and Performance Testing with Test Agents 33
Testing for the Single User 35
Creating Intelligent Test Agents 38
Automated Testing 39
Summary 39

CHAPTER 2
When Application Performance Becomes a Problem 41
Just What Are Criteria? 41
Defining Criteria for Good Web Performance 44
Are the Features Working? 46
Is Performance Acceptable? 46
How Often Does It Fail? 46
Web-Enabled Application Measurement Tools 46
The Web Rubric 47
The Four Tests of Good Performance 49
Components of a Good Test Agent 50
Web-Enabled Application Types 51
The Web-Enabled Application Points System (WAPS) 53
The Web-Enabled Application's Framework 57
The Flapjacks Architecture 57
Adopting Flapjacks and Intelligent Test Agents 59
Building Intelligent Test Agents in a Flapjacks Environment 64
Script Languages and Test Agents 66
Generating Meaningful Data 75
Summary 76

CHAPTER 3
Modeling Tests 77
Modeling User Behavior for Meaningful Test Results 78
Lifecycles, Projects, and Human Nature 80
The Micromax Lifecycle 83
Categorizing Problems 83
Prioritizing Problems 85
Reporting Problems 86
Criteria for Evaluating Problems 86
Considerations for Web-Enabled Application Tests 87
Functionality and Scalability Testing 87
Functional Testing 90

Scalability Testing 91
Testing Modules for Functionality and Scalability 92
Management Styles 99
Service Level Agreements 101
Grid Computing and Intelligent Agents 104
The Road to Easy Setup, Use, and Maintenance 105
Self-Healing Systems 106
Understanding Performance and Scalability Criteria 108
Defining SPC 108
SPC in Action 111
Modeling a User's Goals 115
Test States 117
Using UML and Code Comments to Model Tests 118
Putting the Test Together 119
Summary 120

CHAPTER 4
Java Development and Test Automation Tools 121
The Three Waves 122
Desktop Application Development and Test Automation 122
Client/Server Development and Test Automation 124
Web-Enabled Development and Test Automation 127
Achieving the Troika—the Fourth Wave 131
A Test Automation Lifecycle 133
Summary 137

CHAPTER 5
Bridging from Methodology to Design 139
Searching for Tools to Enable the Troika 140
How to Get TestMaker 144
TestMaker and the Open Source Process 145
Spending Five Minutes with TestMaker 146
Installing TestMaker on a Windows or Linux Computer 147
Running TestMaker 148
Getting to Know the TestMaker Graphic Environment 148
Opening and Running Test Agents 150
Building Agents with the New Agent Wizard 153
Why I Like Jython 162
1. Jython Is Quick 163
2. Data Structures and List Iteration Come for Free 163
3. Dynamic Variables with Automatic Typing 164
4. Functions Are First Class 164

5. Java Integration 165
6. Bean Property Introspection 165
7. Sun Is Adopting Scripting in Java 166
Using Jython to Incorporate JUnit 166
JUnit for Repeatable Tests 166
A JUnit Example 167
JUnit and TestMaker 169
Summary 172

PART II
Components, Interoperability, and Optimization

CHAPTER 6
Design and Test in HTTP/HTML Environments 173
The HTTP/HTML Environment 174
What Usually Goes Wrong 177
Compliance, As in Non, and Browser Caching 177
Invalid Data 179
Session Problems 181
Constructing HTTP Test Agents 181
Hands-On HTTP Communication 184
Understanding Cookies, Sessions, and Redirection 193
Validating Response Data 203
Summary 214

CHAPTER 7
Tuning SOAP and XML Web Services 217
The Web Services Vision 218
XML-RPC for Web Interoperability 221
Where XML-RPC Falls Short 227
Universal Interoperability with SOAP 227
Web Service Scalability Techniques 229
Web Service Interoperability Problems 231
Discovery 232
Definition 232
Request/Response 234
On the Horizon 237
Using TestMaker to Understand WSDL 237
Constructing SOAP Calls 242
Different Types of SOAP Calls 245

Validating Response Data 254
Making It Easier to Write Agents 255
Resources 255
Summary 256

CHAPTER 8
State, Design, and Testing 257
A Question of State 257
Lifecycle for Testing Stateful Systems 261
Techniques to Establish State 263
Preparation and Setup Agents 267
Setup_agent 273
Using Databases to Configure Tests 280
Using Lingo to Make Test Content Close to Real 286
Summary 291

CHAPTER 9
Integrating with .NET Web Services 293
Interoperability and Integration 295
How Is .NET Different? 297
Document-Style Scalability 300
SOAP Headers in .NET 302
WSDL .NET Style 304
A Test Agent for .NET Environments 307
Near Term Considerations 315
Summary 316

CHAPTER 10
Building and Testing Intranets and Secure Environments 319
Getting a Head Start 319
Security by Routing 320
Virtual Private Networks 322
Network Segments and Subnets 323
Transport Security 325
SOAP over SSL 330
.NET Passport Authentication 331
HTTP Basic Authentication 332
SOAP and Security 333

Generating Certificates and KeyStores 336
The Java Keytool 339
Summary 341

CHAPTER 11
A Web Application Framework from Construction to Test 343
The Trading Desk and Intelligent Test Agents 343
Scalability Test Goals 344
System Infrastructure 345
User Archetypes 348
Understanding the Test Requirements 350
Constructing the Test 352
Implementing User Archetypes in Code Modules 356
Implementing the Master Component 365
Setup 366
Run 368
Cleanup 371
Property Files for Test Configuration 371
Implementing the Logging Component 372
Avoiding Test Scalability Problems 374
A First Look at the Results 376
Summary 377

CHAPTER 12
Turning Test Agent Results into Actionable Knowledge 379
What to Expect from Results Analysis 380
Goal: Our New Web Site Needs to Handle Peak Loads of 50 Concurrent Users 381
Goal: The Web Site Registration Page Needs to Work Flawlessly 381
Goal: Customer Requests for Month-End Reports Must Not Slow Down the Order-Entry Service 382
Goal Summary 383
The Big Five Problem Patterns 383
Resource Problems 383
Concurrency Problems 384
Component Problems 386
Contention Problems 387
Crash Recovery Problems 388
Key Factors in Results Analysis 389
Scenarios Where Results Data Misleads 391
The Node Problem 391
The Hidden Error 392

Dileep's Dilemma 393
Diminishing Returns 394
Back to the Stock Trading Example 394
Implementing the Tally Agent 394
Summary 399

PART III
Case Studies: Building Reliable Applications

CHAPTER 13
Concurrency and Scalability in a High-Volume Datacenter 401
Introduction 401
The 2Wire Component Management System 402
Understanding What Happens over the Wire 404
Testing in a Complex Environment 407
The Test Method 411
Test Environment Considerations 414
Constraints in the Test Environment 416
Summary 417

CHAPTER 14
Making the Right Choices for SOAP Scalability 419
Why is SOAP So Popular? 419
SOAP Encoding Styles 421
Simple Object Access Needs Simple Testing 424
Should You Let the Tools Do the Driving? 428
May I Freak Out Now? 428
The Performance Kit 429
How Do I Get the Performance Kit? 430
Installing the Performance Kit 431
Getting Started 432
Some Other Things We Found 445
Summary 446

CHAPTER 15
Multiprotocol Testing in an Email Environment 447
Needs and Benefits 447
Scoping the Project 448

Test Design and Methodology 451
User Archetypes 453
Installation and Configuration 455
From Archetypes to Test Agent Scripts 456
Building Muriel 457
Building Marilyn 461
Result Analysis 476
Summary 477

Index 479

Foreword

The ultimate test of every software construction effort is the user's success using the product to accomplish some practical goal. No matter how much state-of-the-art technology goes into the software, success can only be judged through the lens of the user.

Taking this philosophy to heart, Frank Cohen has created and led a thriving open-source project called TestMaker. It has improved the day-to-day lives of thousands of software programmers, quality assurance technicians, and information technology managers.

Since 1990, my company, Cooper Software, has pioneered the Goal-Directed method of interaction design. This powerful and effective method has met with widespread acceptance in the interaction design community. It is heartening to see how Frank has applied these same methods to software test automation and scalable system design. Frank's application of these methods is especially important in today's environments where J2EE, .NET, and the latest integration technologies are used to build information systems.

The new software testing methods introduced in this book will show you how to form and apply effective online goal-directed design and testing techniques. You begin with your user's goals, follow Frank's recommendations for scalable system design, and end with powerful tests that measure your user's success in achieving them.

As a programmer, quality assurance technician, or IT manager, you have several choices of system design, automated test tools, and techniques. The lessons in this book not only present a coherent methodology, but provide you with immediately useful tools, techniques, and code that will automate the testing of your Web-enabled applications, especially Web Services.

Frank Cohen has been leading entrepreneurial software creation efforts for more than 25 years. With this book you can take advantage of his extensive experience helping users achieve their goals.

—Alan Cooper
Chairman, Cooper Software

Alan is the author of *About Face 2.0: The Essentials of Interaction Design* and *The Inmates Are Running the Asylum*. He is also the father of Visual Basic.

Preface

Now that you have finished learning the basics of programming, testing, and application software development, you are ready for the next step. Web infrastructure (routed networks using open protocols in a grid of inexpensive equipment) is everywhere. And yet, until this book, there was no guide to show how your choices in design, coding, and testing impact the scalability, performance, and functionality of your Web-enabled applications. This book will show you a fast and efficient method to go from basic Java knowledge to building production-worthy, scalable, and high-performing Web-enabled applications.

I wrote this book for software developers, QA technicians, and IT managers working in large corporate IT groups, software development companies, and service providers. It expands on other software development books by going from architectural discussions to showing actual working code that you can use in your own environment. The case studies show real-world practical techniques to make software projects reliable, scalable, and secure.

This book prepares software developers for a laundry list of new APIs, protocols, and tools being packed into the next generation of J2EE, .NET, and open-source systems. While these new software libraries, tools, and techniques are a big move forward for all of us, they push software developers, QA technicians, and IT managers to learn even more technology to turn out complex, highly functional, and interoperable software applications.

This book then gears you up to adopt the next generation of Web protocols, including Web Service Interoperability (WS-I,) Security Assertion Markup Language (SAML), Electronic Business using eXtensible Markup Language (ebXML,) and Liberty Alliance. Tools from technology vendors (for example, BEA WebLogic Server, IBM WebSphere, Sun Java System (formerly Sun One), and Microsoft .NET) are on the move. All this innovation gives pause to a software developer, QA technician, and IT manager. In these environments, each choice of tool, protocol, platform, and technique you make impacts system scalability and reliability.

I wrote this book from my experiences as the "go-to" guy for enterprises that need to test and solve problems in complex interoperating information systems, especially Web Services. This book contains a treasure of knowledge, tips, and techniques and is applied with a solid methodology from the more than 50,000 software developers, QA technicians, and IT managers that participate in my TestMaker open-source project. Details are found at http://www.PushToTest.com, the Web-based community meeting place where ideas on software design, testing, and automation are exchanged every day.

In this book, I describe the architectural choices to build Web-enabled applications in Java and show how each choice impacts scalability and reliability. I show how to test and optimize these systems in your own environment. I describe the need for intelligent test agents in Web-enabled environments, describe a test agent framework with a tutorial on the latest Web-based test techniques, and present TestMaker, my free open-source framework for building intelligent test agents to check Java-based Web software for performance, scalability, and reliability. I present case studies and immediately useful code of how Elsevier Science, 2Wire, Sun Microsystems, and BEA successfully use intelligent test agent technology to build scalable Java applications and ensure confidence in their Web-enabled Java projects.

Inside you will find in-depth discussions of a powerful, proven Web-enabled Java architecture, construction techniques, immediately useful code, and intelligent test agents to check Web-enabled applications for scalability, functionality, and reliability.

- Java and J2EE-based dynamic database-driven Web-enabled architecture
- Integrated applications using SOAP, XML-RPC, .NET, HTTP, HTTPS protocols
- J2EE and .NET Interoperability problems solved

- Performance Kits for developers in BEA WebLogic, IBM WebSphere, SunONE
- eCommerce site architecture and optimization
- Secure Internet services using Public Key Infrastructure (PKI), HTTPS, SSL
- Building with the next-generation security technologies: WS-I, SAML, Liberty
- Avoiding and solving concurrency problems
- Architecture, code, and test agents for J2EE, Java Web Services, P2P, .NET
- Easy-to-understand test scripts using Python/Jython and Java
- Extended architectures include email protocols (IMAP, SMTP, POP3) applications

The PushToTest Web site supports this book with an active community of users. The Web site contains all of the book's source code and applications, ready to be expanded and customized to meet your needs to build reliable and scalable Web-enabled applications in Java.

I hope this helps.

—Frank Cohen

Acknowledgments

"No man is an island," so said the English poet John Donne. In my experience writing this book, I learned that no book is an island either. This book is the culmination of thousands of people's work to improve the way we build and test Web-enabled application software. Each of you provided your feedback, comments, and suggestions. For me the experience has been richly rewarding. I hope it will be rewarding for you too.

There are many people who I would like to specifically highlight for their contributions. My lovely wife, Lorette, supported me on all those sleepless nights and with her thoughts and opinions on how the book should be structured. My children, Jack and Madeline, missed many nights and weekends with me as I disappeared down that rabbit-hole of a home office. Thank you, and I love you more than molasses.

Geoff Lane came up with the original idea to put my test tool, TestMaker, into open-source distribution. He architected the Test Object Oriented Library (TOOL) in TestMaker and continues to provide his brilliant thought to the project. Because of his technical editing I covered new subjects and expanded coverage throughout the book.

Todd Bradfute has become my major collaborator in TestNetwork, a commercial version of TestMaker. His feedback and designs for a distributed test environment have helped this book greatly.

Darin MacBeath of Elsevier found the problems with SOAP RPC scalability that are highlighted in this book. To my knowledge Darin's designs for a

next-generation content publishing engine that uses SOAP as a means to publish APIs was the first SOAP-based Web service in production.

Alan Cooper taught me the Goal-Directed interaction-design method he invented and encouraged me to write this book.

For helping me deliver this book: Marc Farley (mentor), Carole McClendon (agent), Karen McLean (editor), and Jeff Mason (coxswain). The technical editors provided excellent feedback: Christian J. Hessler, Eben Hewitt, and John Mueller. The readers provided their insight: Paul Gerken and Philbert Canon.

For encouraging me to write this book, my late Aunt Betsy Hilbert, herself published many times over.

For their support of TestMaker I thank: Eric Brown, Farooq Kahn, Ronald Tschaler, Linda Bissum, and Bao Nguyen at 2Wire; Dileep Kumar, Madhu Konda, Rohit Valia, Robin Smith, and Deborah Andrade of the SunONE team at Sun Microsystems; Phil Goodwin and Jeff Sutor on the Java engineering team at Sun Microsystems; Michael Smith, Eric Stahl, Scott Regan, Sam Ramji, Samir Kothari, Michael Bamford, Michael Smith, and Andrew Sliwkowski at BEA Systems; Karen Lee, Deborah Magid, Rod Smith, Rawn Shah, and John Swanson at IBM. PJ Murray and Robert Schneider at CapeClear; Robert Scoble at Userland; Rossana Muriel and Ernest Semerda at AMP.

Many individuals helped to shape this book, including Raymond R. Leiter, Morris Hoodye, Keyton Weissinger, Todd Green, Jim Schacterle, Brigitte Gwin, Tom Clements, John Duhring, Norman Cohen, Danny Berlin, Alex Sarellano, and a good many others who I have foolishly not mentioned here.

Finally, thank you to you for your interest in this book. Buying this book puts food on my family table and keeps me going to improve my tools, techniques, and methods. I appreciate your interest and hope the experience is rewarding for you personally and professionally.

You are about to hear from me, I would like to hear from you. Please write or email me at fcohen@pushtotest.com and tell me what you thought about this book, and about testing and building scalable Web-enabled applications in general. Let me know your contact information (including email address) and I will keep you informed about my current and future work, new products and services, and new books and articles.

Also from Frank Cohen

1

The Forces at Work Affecting Your Web-Enabled Software

In my experience, more than just the way you write code impacts Web-enabled application functionality, scalability, and performance. Here are three forces working against your software development efforts:

- Working within the software development, quality, and information technology efforts at your company impacts functionality, scalability, and performance. High-quality interoperable Java code is developed in processes among groups of individuals. The first part of this book describes the organizational and social issues that impact functionality, scalability, and performance. You will then immediately learn useful new processes and techniques to measure and improve scalability and performance.
- The design and architecture choices (server software, protocols, schemas, platforms, hardware, routing) you make impacts functionality, scalability, and performance. The second part of this book describes the technology behind interoperating and scalable Web-enabled applications from a Java developer's perspective.
- The way you test and monitor system and software modules and components impacts functionality, scalability, and reliability. The third part of this book presents case studies and

immediately useful intelligent test agent code to check Web-enabled applications for functionality, scalability, performance, and reliability.

This chapter begins with a practical definition of Web-enabled applications and the Web infrastructure they run on. This chapter then tackles the difficult and thorny behavioral problems found in many companies that make delivering interoperable, high-quality Web-enabled applications in Java truly difficult. This chapter then gives a brief history of software development practices over the years to show you how many of the software development, QA, and IT practices in place today came to be. The chapter then describes the architectural and software development practices that impact scalability, performance, and reliability in Java software. Finally, this chapter sets the stage for the rest of the book by introducing a new way of testing your application using intelligent test agents, a new concept presented throughout this book.

The First Three Axioms

I remember sitting in front of my first microcomputer. It was 1977 and I was in high school. I typed on the keyboard:

```
print "I hope Maggie Movius likes me."
```

Sure enough, the BASIC language interpreter took my one-line program and displayed on the terminal:

```
>I hope Maggie Movius likes me.
```

What a marvel! It seemed so simple, so understandable, so... hard to explain to anyone that had not tried it. I wondered how much I could accomplish with a microcomputer.

The years rolled by. I watched and participated in forming the software development community. With the community came an emphasis on software development practices. Software moved up from being an optional feature in a computing box to a necessary enabling component of the U.S. economy.

The software development community developed a means to rapidly share good ideas on how to build software, and how to dispute and put down ideas that were bad. For example, threads and streams = good and proprietary communication protocols = bad.

Not long after, I noticed that more complicated software, such as a Web server software package, consists of the contributions of many software developers. In the commercial software space, teams of software developers meant paychecks. And with paychecks came management, software product lifecycles, quality assurance, version control, product launch schedules, and code reviews.

In an effort to return to a simple development example, while working in a development team of a commercial software company, I typed out this simple example program:

```
printf ("I hope Maggie Movius likes me.");
```

I compiled the code and checked it in to a source code control system. About an hour later, a QA technician sent me an email:

```
Hi Frank: Nice to have you on-board. I saw your first check-
in. You should know that your code is missing the copyright
notice for the company and it does not meet with our coding
standards since you used printf instead of printing to an
output window in our standard library. -Norman
```

The experience taught me these axioms of software development:

- Even though a program contains only a single line of code, it needs to be maintained.
- Every piece of software interoperates with other software.
- There is more to delivering high-quality software than writing code.

Zoom forward to present. As the principal architect of TestMaker, a popular free open-source test automation utility, I find these axioms help me to deliver interoperable, Web-enabled, production-ready Java software, quickly and on a budget. In this book, I show you more axioms; present practical ready-to-use techniques to build, code, and test high-quality software applications; and provide test code that you can use in your own environment today.

Web-Enabled Applications, the New Frontier

Software development goes through phases of development and digestion. In the development phase everything is up for grabs. Software developers experiment with software architectures, especially with the location of application business logic and presentation code. Presentation code handles windows, mice, keyboard, and other user interactions. Business logic is the instructions that define the behavior and operation of an application. In the digestion phase, software developers try to conform to the current widely accepted best practices.

Figure 1–1 shows three cycles of the phases of development and digestion. The first-generation software architecture built the presentation and business logic on a single system. In the second generation, client/server architecture brought back the large and centrally controlled datacenter so familiar in the 1960s when mainframes ruled the information world. In client/server architecture the desktop system is a "dumb" terminal that only needs to display the data provided by the server. The early Web was modeled after client/server architecture. The browser makes requests to a server and renders the received HTML codes to the screen.

As browsers improved in functionality—applets, JavaScript, ActiveX components, and DHTML were introduced—some systems included business logic on the desktop side. However, the majority of functions remained on the server.

Figure 1–2 shows what I expect to come next. The "grid" generation enables an application to host business logic modules on the desktop or server.

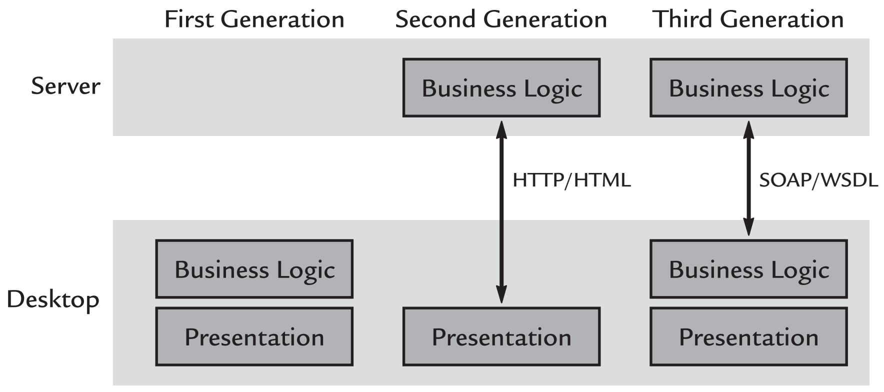


Figure 1–1 The first three generations of software development brought us from desktop-bound application, to client/server, and then to the early Web.

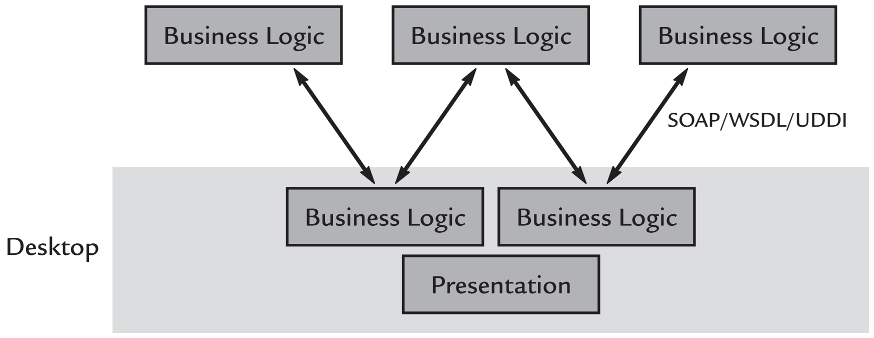


Figure 1–2 The coming generation uses Web Service and peer-to-peer (P2P) technologies running on existing Web infrastructures.

The modules discover each other using Web service and peer-to-peer (P2P) technologies. In this generation, multiple copies of the business logic modules may run in a grid of datacenters to allow failover, dynamic routing, and functional specialization. These architectures run on the Web infrastructure already in place at most enterprises. On the way to the "grid" generation we will watch our Web infrastructure grow to support *Web-enabled applications*.

Web-enabled applications are interoperating systems of software implementations that use open-standard protocols to communicate over a Web infrastructure. They started in very simple terms. A browser makes a request to a server for a document. Everyone rapidly realized how inexpensive it was to operate a network and Web server. Consequently, today it may seem like everyone does. By 2003, the NetCraft survey of Web servers received responses from 40,936,076 sites—and that number is just Web sites. Enterprising software developers use the same Web infrastructure for hundreds of other types of applications.

Consider the typical enterprise infrastructure today. Many of the applications an enterprise infrastructure supports are shown in Figure 1–3.

The functions shown in Figure 1–3 are typical of the services provided using a Web infrastructure at a typical company today. The following describes these services in detail:

- **Sign-in function** enables an individual employee or customers to identify themselves to gain access to private information or privileged operations. The function uses a browser over an open HTTP connection. The server response is simple HTML codes indicating success or failure. (Most, but certainly not all, Web

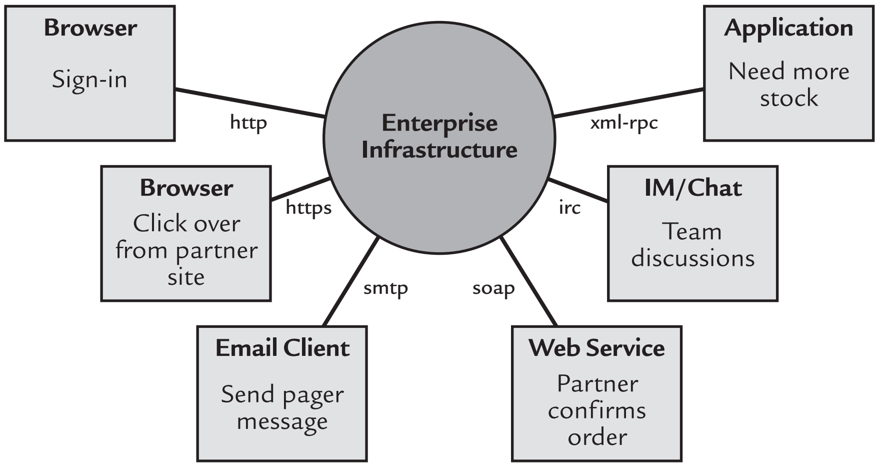


Figure 1–3 Modern Web infrastructures support a wide variety of Web-enabled applications and functions.

applications now use the secure HTTPS connection when dealing with sign in functions.)

- **Click over from partner site** enables a user that already identified himself while using a partner's Web site to view private information or access privileged operations without needing to sign in a second time. The partner Web site acts as the authentication service to accept or deny the user. The enterprise infrastructure accepts the authentication credential from the partner Web site. The server response is an HTTP browser redirect command to point the user's browser to a URL within your site, or redirect the user back to the partner Web site.
- **Send pager message** enables email forwarding to an employee's pager. In this function, the enterprise infrastructure receives standard Internet email messages. The function operates as an SMTP email service. When it receives an email message, it looks up the employee's pager number from a directory and forwards the email message to the pager gateway. The service responds with an email message confirming the page is sent.
- **Partner confirms order** enables a partner's automated order entry system to confirm the details of an order that is in your fulfillment system. The request comes as a SOAP-encoded Web

Service request. The response contains delivery information for the order in XML format.

- **Team discussions** are facilitated using the infrastructure's Internet Relay Chat (IRC) service. The function uses IRC client software to marshal and distribute requests and responses. Communication between team members contains valuable knowledge, so the IRC message traffic is recorded, indexed by a search engine service, and displayed through a Web browser interface.
- **Need more stock** enables non-browser-based applications to communicate out-of-stock conditions to an order entry service. Imagine a store like Wal-Mart with sensors built into the store shelves that sense when more socks are needed. The function requests more socks by using XML-RPC protocols to make an XML-encoded request over an HTTP protocol to the order and inventory service. The response contains an acknowledgment and confirmation number.

This book shows how to build and test systems that provide all of these services. Additionally, this book prepares you for the coming wave of services and protocols that run on the same Web-enabled infrastructure. Before we go on, look at Figure 1–4 to see what happens in a back-end system that handles all of these functions.

The back-end systems that enable the enterprise infrastructure to provide all of the services in Figure 1–3 also use standard Web technology, including:

- **Authentication Authority** is a service that maps users to resources. For example, an authentication authority determines if a user may write a document to a printer device.
- **Application** is a service that implements the unique business processes and workflows for a company. Applications are usually implemented in two parts: a set of Java servlets to handle the HTTP communication protocol with the browser and to initiate a workflow and a set of Java or Java Enterprise Bean in J2EE objects that implements the workflow. For example, the sign-in request from Figure 1–3 comes to the application as an HTTP request from a browser that triggers a servlet to start the sign-in process in a Java bean. The bean makes an authorization

assertion to the Authentication Authority using the Security Assertion Markup Language (SAML) protocol to ask if the user is authorized to sign in. (Of course, application service is not limited to Java. Developers have many choices of operating system and platform to implement the business logic of an application, including Java, .NET, C, and C++.)

- **Network Operating System** (NOS) controls the network layer access to devices and services. The NOS is usually Windows, Unix, Linux, Macintosh, or other operating system that provides a software interface to network resources, hardware resources, and a simple directory service.
- **Instant Message Server** (IM) provides protocol-level support for Internet Relay Chat (IRC) service. In the team discussion example from Figure 1–3, a group of developers use IRC to communicate and coordinate development of a new software module. The conversation includes information private to the company. Requests to join an IRC chatroom initiate a workflow that uses the Lightweight Directory Access Protocol (LDAP) to make sure the user is in the Network Operating System directory.

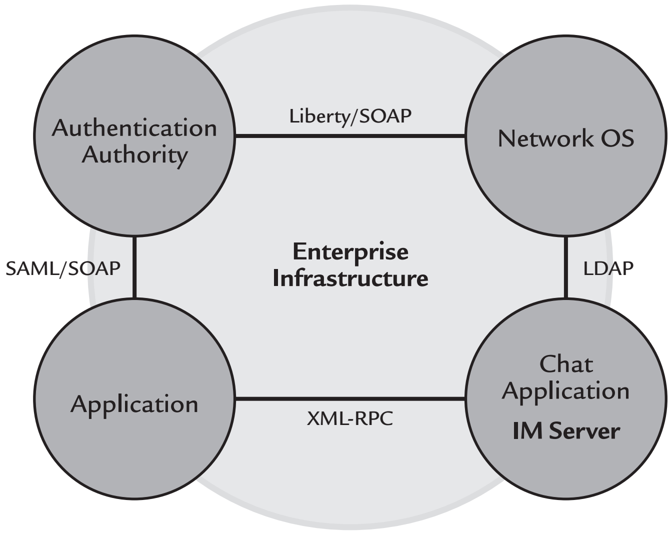


Figure 1–4 The back-end systems use the same Web infrastructure to connect many individual systems to service the variety of applications making requests.

The systems and protocols described in Figure 1–4 are by no means thorough. In fact, looking at the average enterprise back-end system will show you dozens of other systems and protocols. The glue that binds all these systems together and delivers the widest audience of users is a Web infrastructure.

What is driving all this development on top of Web infrastructures? From my perspective, we are living at a time of "firsts." Consider the following:

- This is the first time in the information technology (IT) industry that software developers, QA technicians, and IT managers agree on interoperability architecture. We can expect Web infrastructures to be already installed at every company.
- This is the first time that developers, QA technicians, and IT managers agree that security needs to be federated. The company we work for, or the group we participate in, now exists in partnership with other companies and groups. Consequently, our systems need to share authorization, identity management, and directory services. We need to provide federations of Web sites where a sign-in to any one Web site signs us in to all Web sites within the federation.
- This is the first time that developers, QA technicians, and IT managers agree that scalability in every application is critical to achieving our goals. From a single function in a company to the entire world economy, everything relies on Web-enabled applications. When they don't work, or work slowly, everything grinds to a halt.
- This is the first time that government regulation is part of our software application requirements. For example, The U.S. Congress passed the Health Insurance Portability and Accountability Act (HIPAA) in 1996. HIPAA is expected to have a major impact on every healthcare organization in the U.S. HIPAA requires strict controls on access to patient data and requires that patient data be moveable when patients move to different healthcare providers. HIPAA is a precursor for the future of information system's best practices crammed into a government regulation.
- This is the first time that interoperability tools and technology is delivered with integrated development environment (IDE) tools. In the past, engineers wanting to work with CORBA would turn to an external CORBA toolset to define interface

definitions. Now the most popular IDEs come with CORBA, Web Service, RMI, COM+, Message Queue, and Database/ Dataset Access tools built-in.

- Finally, this is the first time that Java and other interoperable development environments and open source projects have reduced operating system and platform dependencies to a point where, in many cases, it really does not matter any more. In a Web-enabled application environment, all the OS and platforms have reached near feature parity. Software developers are being courted by OS and platform providers to recommend products, tools, and technologies, that marketing used to be limited to system administrators and IT managers.

We've lurched the computer software industry forward with these "firsts." The fear that used to inhibit software developers from working on distributed, interoperating systems is gone. A flood of new protocols, tools, and techniques is upon us. For example, the SAML defines a protocol to facilitate single sign-on, Liberty Alliance enables developers to build federated security models into Web applications, WS-I standardizes a common set of Web Services programming interfaces and protocols, and BPEL4WS defines a way to express and implement business workflows in Web Service applications. And there are many more on the way. Part II of this book presents many of these new protocols and gives you a way to approach good design and test strategies for those protocols that will appear in the near future. All of these new protocols are built to run on Web infrastructures and generally are easy to learn. The wild ride we are about to have will lead the world to higher productivity and efficiency.

The next section tackles the difficult and thorny behavioral problems found in many companies that makes delivering interoperable, high-quality Web-enabled applications in Java truly difficult.

Why Writing High-Quality Software Today Is Hard

The computing world offers the largest-ever selection of software development tools, techniques, hardware, and knowledge to build interoperable, high-quality Java software applications. And yet, using the finished application is often an incredibly frustrating experience for users. While there is much that we, as Java software developers, can do to improve the software code we write, our participation in the larger business effort has an equal impact on users achieving their goals, and avoiding frustration. I would bet that you have seen and experienced one or more of the following five problems impacting software development efforts:

- The Myth of Version 2.0 Solidity
- Management's Quest for Grail
- Trying for Homogeneity When Heterogeneity Rules
- The Language of Bugs
- The Evil Twin Vice Presidents Problem

Each one of these problems has its impact on your software development effort, but is apart from actually designing or writing code. Let's take on each one of these individually.

The Myth of Version 2.0 Solidity

Has this happened to you? You rush toward a big new software release. You pin all your hopes that the next major version will be rock solid, perform well, fix all of the existing problems, and give users a successful experience. Additionally, you anticipate the new software will prove itself to your management, which in turn will get you a raise and let you go on vacation. Back to reality...

Writing code is hard. In a modern Java development shop, your code modules make up a larger system and at any time the other modules may change. Yours is a world where everything is a moving target. Add to the mix that there is always something new to learn and that you may not like the way the developer sitting next to you smells, but your module depends on theirs!

In software development, there is always the big question of making lots of little changes versus making a few big changes. Agile programming techniques favor many small implementations over a major overhaul. Management that believes in *Version 2.0 Solidity* chooses the major overhaul. In my experience, choosing the major overhaul is a terrible choice unless you have really thought through the implications. More often than not good development teams implode from the stress of accomplishing the "big release." It is just not worth it. The finished software is bound to have bugs and problems that need fixing immediately but software developers are weeks away from rebuilding their energy to handle a maintenance release. If they don't get that "decompression time" then they quit or do something stupid, which gets them fired.

The best software development comes in waves. Inspiration overtakes an individual software developer and the developer codes fast and furiously until the inspired vision appears in a block of code. The rest of the development team agrees with the design, the developer integrates the new code into the existing code-base, and users appreciate the new or reworked functions. From there, the code is refined until it is later replaced by a redesign from another inspired developer.

Agile programming techniques embrace this idea that inspired software development comes in waves. At the end of the day, you will need to decide if agile programming works for you. I have spoken to software developers that like the inspired wave technique, but not the rest of agile programming. I recommend you do what works best for yourself.

In the continuum of software development—from making minor changes to maintain existing code, to development of a major new release that replaces an existing code-base—the "inspired wave" works well for the middle efforts, more than maintenance but not an entire rewrite.

Management's Quest for Grail

In the eco-system of software development, engineering management participates by directing resources and measuring progress against milestones and checkpoints. Many managers believe their mandate includes choosing how software is developed. A manager may look at a new development technique and insist on its adoption. This poses a problem to the software developer whose problem-solving style and development strengths run counter to management's mandated development techniques.

In my experience, agile programming (also known as Extreme or XP programming, and its supporters such as SCRUM, see http://www.control-chaos.com/) often looks like a "silver bullet" to management. Management's thinking goes: Delivering the 2.0 project was a giant mess; we should have instead used agile programming techniques to issue a bunch of little releases. In reality, there is no linkage between agile programming and the "Version 2.0 Solidity" myth, though it looks like a solution to management.

There are parts of agile programming that greatly benefit a software development organization. For example, agile programmers believe in taking lots of little bites and then testing the bites of new code as they are written. That might come in handy when dealing with "Version 2.0 Solidity" problems. The problem is that management may mandate agile programming for all developers.

Software developers have different problem-solving styles and some styles work against agile programming techniques. Understanding your own strengths and problem-solving style is important to delivering successful software. From my experience, programmers can be categorized by problem-solving style and software development strengths. Figure 1–5 illustrates the types of programmers and their relative sizes.

From the start, you might wonder why users and super-users are included in Figure 1–5. In my opinion, every computer user is a programmer. How often have you seen PostIt notes with a user's favorite tip or technique to use an application stuck to the side of a monitor? The PostIt note is programming.

Figure 1–5 shows that each type of programmer has a different problem-solving style. Users, the largest group, use procedural programming styles to solve problems. They work within the application's existing structure to accomplish tasks. Superusers are procedural problem solvers too, however, they typically use more than one application and so if a problem is unsolvable in one application they will seek a solution in a second application. Administrators are also procedural problem solvers, however, they have access to more functions than users and superusers and may even be in contact with the writers of the code. That access gives them more insight to solve problems.

Procedural programmers and script writers are entrepreneurial problem solvers. It is fine with a script writer to have multiple solutions existing at once. The more solutions they are involved with, the happier they are. Code reuse, efficiency, and maintenance are secondary concerns to solving problems immediately. Script writers work within an environment where the

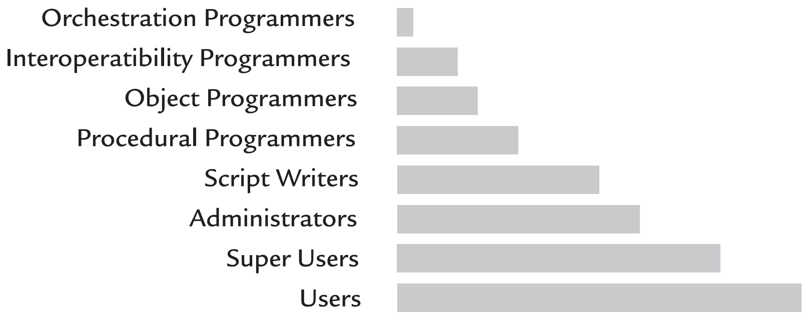


Figure 1–5 This shows the relative sizes of the different types of problem-solving styles among programmers.

overall software architecture is implemented by the software development tool or framework they are using. For example, Visual Basic users visually lay out an application and the development environment asks the developer to fill in the execution code for each visual button. The development environment writes all the other code to initiate, run, and close down the application. Procedural programmers also work in environments where someone else does much of the architecture and they seek to simply code a function now.

Object programmers have got the reuse religion: Software code should only be written if it completes a missing function within the overall system. What's more, the choices one makes during software design must make it easy for anyone else on the team to use the software's functions, in whatever context happens. Object programming is the high opera of software development. Every object design choice is elevated to a hugely dramatic level because even the smallest object counts toward the overall system's functionality and performance.

Interoperability programmers bring back that entrepreneurial spirit. They view all systems from the perimeters. They design application software that occasionally makes functional calls through the application programming interfaces (APIs) offered by a Web-enabled application, especially an XML Web Service. However, they are different from object programmers as the APIs will normally offer flat and non-object-oriented method calls.

Finally, orchestration programmers solve programming problems by designing the functionality they need first and then defining the business workflow to achieve the goal. The code an orchestration programmer writes has no graphical user interface, instead all the application's decisions are programmed into a set of preferences that tells the application how to walk through a workflow to achieve a desired response. For example, the loan application process at a bank is a workflow and an orchestration programmer implements software that steps the loan through its many approval steps.

Imagine what happens when management implements an object-oriented design process among a group of procedural programmers. Imagine what happens when management decides to use a business orchestration system for their development team of script programmers. Understanding the problem-solving style of the software developers is key to delivering successful software applications.

Another way of looking at the types of software programmer problem-solving styles is shown in Figure 1–6.

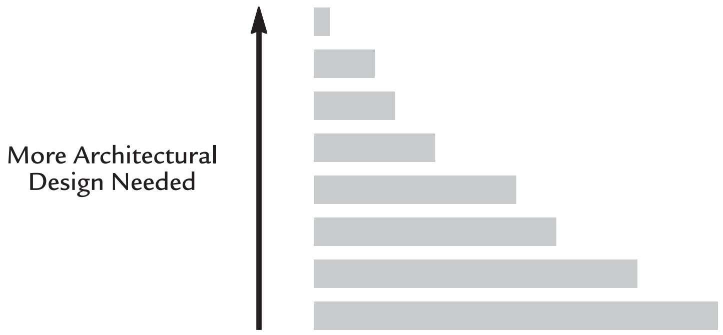


Figure 1–6 For the different types of problem-solving techniques in computer programmers from Figure 1–5, it turns out that the smaller the group, the more need there is for architectural design of the software under development.

Figure 1–6 shows the relative sizes of the subgroups of software programmers by problem-solving type. For example, in general, there are many more procedural programmers than object-oriented programmers. This has an impact on the way we design successful software development tools, platforms, and operating systems. For example, if our intention is to attract a large audience of system administrators who will develop or maintain application software, then our design must match their problem-solving style by making it easy to understand the major modules, communication between modules, external access protocols, naming methodologies for objects and variables, and resources to be used in a software application. On the other hand, if our intention is to attract orchestration programmers, then our design must focus on how to connect a set of available Web services.

Understanding your own strengths and problem-solving style is key to understanding how management's decisions impact your software development efforts.

Trying for Homogeneity When Heterogeneity Rules

My high school chemistry teacher enjoyed teasing his students with paradoxes. For example, he would ask, "If there was a universal solvent, then what would you keep it in?" I think of him when I meet software developers, QA technicians, and IT managers that tell me their company is trying to standardize their infrastructure on a single platform. Ah, the quest for homogeneity. It's clearly a paradox of the first order.

The paradox is that to have a homogenous infrastructure means you need to remove anything foreign. That never happens. So instead heterogeneity rules the day! Everything is built on top of whatever preceded it. It is messy. It is difficult to maintain. It is brittle—any one break causes multiple breaks elsewhere. Lastly, eventually the oldest systems become archaic when no one is left around that knows how to do maintenance.

That is not to say that efforts for homogeneity in the world are on the retreat. Hardly. Instead this is a battle we must fight frequently. Because—as readers of this book—we are Java developers, QA technicians, and IT managers, we regularly have to solve functionality, performance, and scalability problems in heterogeneous systems. So I offer these points to keep in mind when dealing with an effort to make your heterogeneous infrastructure homogenous:

- Plan for patches to your system, to system components, and to legacy systems that no one has looked at for years. In a heterogeneous environment, patches are more likely the only possibility unless the company will consider replacing a system.
- Design for brittleness in the systems providing supporting service to your application. At every point in your application when resources are touched, handled, or managed, make sure your code can handle the scenario where the resource is not available.
- Accept the moving-target nature of the datacenter. Your application needs to play with other applications well. This is especially true when new applications appear in the datacenter without much coordination among the teams in your company.
- Provide client-side system-level tests for your more complicated applications. In addition to building a unit test, also provide a client-side test that drives the entire solution, rather than just the individual functions you delivered. The system-level tests will show future engineers when they have broken something and the QA technicians and IT managers will have a tool to diagnose when your module or application fails. Later in this chapter I introduce a new technique for building client-side system-level tests.

By now you probably understand how these problems are linked together. As a collection, all five problems conspire to make software development messy, politically challenging, and unpredictable. The following are the remaining two problems that impact successful software development.

The Language of Bugs

In my experience, most companies, by default, choose a bug list as the common language spoken between the software developers, QA technicians, and IT managers. Rather than asking if the financial advisors using a stock trading application are meeting their goals, the question comes out like this: Is Bug 38283 fixed?

When writing or maintaining a software application, the language you choose will impact your ability to deliver successful software. This is covered later in this chapter in a discussion of user archetypes and intelligent test agents. For now, consider an example case where you write a new module for a software application. Which of the following test strategies would you choose?

1. Test the code by sending it a single sample value. If it passes the test then you can check-in the code, and hand it off to QA and IT groups.
2. Test the code by trying to operate it in the widest possible way to emulate the many varied ways all the different users will use the module.
3. Test the module by sending it invalid data. If it fails, then it's ready for a hand-off to QA.

Have you noticed that something is missing? What happened to the *single* user? You know … the single user that uses the application to achieve their goals. They walk through the application, operating many functions in succession, each function operates a group of modules, until they hit the right combination of functions and … BOOM! The application falls to its knees.

Most developers I meet believe the three choices above are the *only* choices. They think they must either test for the fringe cases of invalid data or test widely for every possible case. I have even seen developers try to split the difference by testing right down the middle.

Testing to the middle makes large assumptions of how the aggregate group of users will use the Web-enabled application and the steps they will take as a group to accomplish their common goal. In reality, the form and function of modern software applications makes it possible for users to choose their own path through the functions in an application. Each user has a personal goal and method of using the application. But how do you describe such a user and communicate to the others in your team that you want to test for success in this one user?

I recommend building software by first building a test agent modeled after a single user. Choose just one user, understand their goals, and learn what steps they expect to use. Then model a test against the single user. The better understood that individual user's needs are, the more valuable your test agent will be.

Some developers have taken the archetypal user method to heart. They name their archetypes, describe their background and habits. They give depth to the archetype so the rest of the development team can understand the test agent better. Goal-driven user archetypal testing provides a hugely effective way to communicate between developers, QA, and IT.

Next, we discuss the last of the top five problems that impact your ability to deliver successful software applications.

The Evil Twin Vice Presidents Problem

For the final problem on our top five list of problems that impact your ability to deliver successful software, consider the company where management responsibility for software development and quality assurance is in one group and responsibility for running the datacenter is in the other group. This situation seems to happen all the time. Yet, it sets the scenario for sniping between the software developers, QA technicians, and IT managers.

If there was ever a need for another telethon it would be to stamp out this problem. And in our time. When it exists, everything slows down, schedules go unmet, and feelings get hurt enough for people to quit.

One solution I have helped implement in this kind of situation provides a framework for software developers to conduct unit tests that may then be leveraged by QA technicians for scalability and performance tests. IT managers, to prove Quality-of-Service reports, can then leverage the same unit tests. (You will find me coming back to this theme many times throughout this book. As you will see in Chapter 2, I call the solution *The Troika*.)

In summary, this section presented a list of the top five problems that impact your ability to deliver successful software applications. The problems are mostly behavioral and do not involve actually writing the code of an application. While it may appear that these problems should not concern you, this book tackles these problems head-on with a new testing methodology, new set of test tools and framework, and a rich history of software development to draw from to help you succeed, even if your company is suffering through any or all of these top five problems.

A Concise History of Software Development

Modern Web-enabled applications are built differently than software developed in the past. As we found earlier in this chapter, even a single-line program will interoperate with libraries of code developed by people you will never meet. If you visited International Telephone and Telegraph (ITT) in the 1970s, you would have found that an ITT engineer wrote every piece of software they ran in their mainframe datacenter. Zoom forward to the present and you will find that applications are woven together of libraries from diverse and often unconnected code. So how did we get here?

In the early days, computer software was easier to develop, because it did less. Computer software was easier to test, because the company owned every part of the infrastructure. Computer software was easier to design for scalability, because the end users were the known and finite group of company employees. Computer software may have been easy to develop, but it was often hard to use. Some projects may have improved business efficiency, but they were nightmares to use, and worse to test.

The modern software development age was born when users, developers, testers, and management adopted Web environments. In a Web-enabled environment, an application is computer software that offers interoperability and connectivity features to other systems. Even a pocket calculator, like a personal digital assistant (PDA), is expected to connect to the Internet. This book refers to these interoperating and connected software applications as Web-enabled applications. The devices used may be pagers, mobile phones, laptops, game consoles, dedicated devices, desktop computers, and servers. If the software inside uses Hypertext Transfer Protocol (HTTP), Simple Object Access Protocol (SOAP), Simple Mail Transfer Protocol (SMTP), or any other Internet protocol, the software may be considered a Web-enabled application.

Developers have reached entirely new levels of productivity by building Web-enabled applications. Business managers see new software initiatives as the quickest and most efficient way to increase productivity, and users love all the new access and functions available to them for the first time. With the right choice of software testing tools, techniques, and methods, Web-enabled application testers and system managers deliver the highest quality, most reliable, and most available computer software ever developed.

This book shows the experiences of developers, QA technicians, and IT managers who have found that delivering high-quality Web-enabled applications requires intelligent test agents, new methods for analyzing scalability and performance, and a good understanding of how the behavioral issues and personalities in a development team affect Web-enabled application testing and monitoring. The appropriate techniques identify a Web-enabled application's scalability and performance index, which are used to plan an appropriately sized data center and monitor Web-enabled applications to achieve a service-level agreement.

A major portion of this book shows what happens when traditional, last-generation software testing and monitoring tools and techniques are used with Web-enabled applications. Sometimes the results are meaningless—or worse, the results are misleading. The new software test techniques presented here help developers prioritize their efforts to maximize the user's ability to achieve important goals. The book presents new test tools and easy-to-follow criteria for determining a Web-enabled application's health. Finally, this book presents new data center frameworks and an open-source test automation tool that provides easy and efficient ways to build Web-enabled applications that scale and perform in real-world production environments today.

Web-Enabled Applications

Decades ago, software developers were concerned with writing software applications only after the electronics, bandwidth, connectivity, and other hardware devices were implemented. A 14.4 KB modem connection might fail midstream. The system you were connecting to might have only partially implemented the Kermit protocol. The 10-megabyte hard disk drive on the server may have filled up and crashed the operating system. (Do these archaic technologies take you back?) These early rickety systems had constant failures that the software application had to handle.

We learned that when the underlying platform is rickety, so is a software developer's concentration. Instead of concentrating on the form, flow, and logic of an application, the developer worries about writing lots of error-handling code. In the early days, a well-written program may have contained 300 lines of code, but only 50 lines would have performed the actual business logic of the application. The rest handled errors. Today the underlying platform is well defined and deployed. Software projects have an Internet connectivity strategy, an object-oriented programming language, a programmatic interface, and a set of user interaction expectations. And while new protocols and APIs might add new complexity back in, it is common in today's applications to see 300 lines of business logic implementing code, and only 50 lines of error handling code.

Web-enabled applications benefit network and IT managers by providing more flexibility to deploy and manage a system. Scalability is achieved by using many small, interconnected servers over large-scale monolithic mainframes. Deployment, vendor management, and upgrade maintenance are much easier in a Web-enabled application environment.

Web-enabled applications have also enabled improvements in the tools software developers use to write and maintain applications. Today, rather than using individual tools to develop specific applications, developers employ the IDE. IDEs battle for market dominance by incorporating many individual tools and functions. At a minimum, an IDE today must offer a code editor with contextual help, a compiler, a debugger, a client to a source code revision system, a deployment packager, and templates to allow the programmer to write new applications, modules, and classes instantly.

IDEs provide a place to write your code with the editor, compile it into runable code, and then use the application in a debugger environment. Debuggers execute the program line-by-line or until a certain condition or breakpoint is reached and then show the variables and data structures in the live code. Compared to the old days, IDEs are a godsend.

Unfortunately, IDEs and debuggers have not yet caught up with Web-enabled applications running on a remote server. To test these systems often requires the business logic in the application to reach certain states before the software shows meaningful debugging information to the developer. To solve these problems, test paradigms have appeared over the years, including the Brute Force Testing Method, Mainframe Software Development Lifecycle, Desktop Software Development Lifecycle, and Internet Software Development Lifecycle.

Test Paradigms and Lifecycle Processes

Modern software is developed in lifecycle processes. Software is written, tested, and deployed. Then the cycle begins again for maintenance. Software development lifecycle processes brought about many changing test methods and paradigms.

By the time desktop computers emerged as a standard, the Brute Force Testing Method was firmly entrenched. The Brute Force method employs many engineers sitting in front of desktop computer systems performing the following tasks:

1. Operate the keyboard and mouse.
2. Use a program like UNIX tail to view a log file.
3. Write debugging code to save the application's internal values and states to the log file.
4. Examine the logs.
5. Repeat 1,000 times a day.

Eventually, the testers run out of patience, friends and family will no longer test the application (even as a favor), and that one giant bug that happens only once every 3 hours won't go away. The Brute Force Testing Method rapidly reaches a point of diminishing returns. After a few years of this, the software community realized the existing software development lifecycle needed to change.

To see how software development lifecycles changed over the past three decades, let's consider what happened as new hardware, development tools, and connectivity improvements came on the scene. Back in the days of mainframes, software tests were accomplished on the actual production mainframe using simulations and test data. I coin the software development lifecycle that emerged as the *Mainframe Software Development Lifecycle*. The steps in this lifecycle look like this:

1. Specify the program, subsystems, and terminal interface menus.
2. Write the initial documentation.
3. Write the major modules.
4. Simulate modules running on a system.
5. Analyze the simulation results for anomalies.
6. Code the minor modules.
7. Run the system on the production system with test data.
8. Analyze the results.

9. Fix any problems with the implementation.
10. Finish the documentation.
11. Declare the final candidate, and then ship software to the customer.

Unfortunately, it turns out that the tests were only as good as the simulations. Maintaining the software was equally unsure because new maintenance versions were also tested using simulations. These tests also brought about the need for the systems analyst—it turned out that the systems analyst became yet another person to get between the user goal and the engineer implementing a solution.

The systems analyst looked at software systematically and often missed or ignored the usability and performance problems baked into the software. As an example, in the early 1980s, airport airline reservation computer systems could not look up frequent-flier mileage information while agents were looking at a passenger's reservation. The reservation application correctly displayed the itinerary, but the agent had to exit the application and start the frequent-flier application to access this information. The reservation application and frequent-flier application passed the systems analyst criteria for quality and performance; however, the airline customer did not achieve his goal of moving through the airport quickly.

By the late 1980s, the burgeoning PC industry made possible a software publishing market for desktop application software. These were the heady days of Norton Utilities, dBase II, and WordStar. *The Desktop Software Development Lifecycle* emerged to build personal computer software:

1. Specify the program from a feature list.
2. Write the software application.
3. Unit test the application.
4. Fix the problems found in the unit test.
5. Quality assurance engineers test the application (alpha testing).
6. Fix the problems found.
7. External beta testers test the application.
8. Fix the problems found.
9. Ship the application to customers.
10. Collect bug reports for the maintenance cycle.

By the 1990s, software QA professionals became part of the landscape and managed the Desktop lifecycle.

When the Desktop lifecycle works well, customers are treated to new software releases no sooner than every six months. During that release time, the software publisher chooses which bugs will be fixed and works toward the next lifecycle. Many users encounter bugs serious enough to ask the software publisher for a more rapid maintenance release. The Desktop lifecycle is too rigid to accommodate shortened release lifecycles. Even small maintenance releases would shave only a few weeks from a six-month lifecycle. Additionally, maintenance releases take the development team's resources away from working on new features.

When software publishers began to be more efficient at managing Desktop lifecycles, the distribution and sales channels pushed back. Apple Computer was able to develop maintenance and new feature releases in less than six months; however, Apple's major product distributors asked the company for new product releases only every nine months. The market delivery system needs a rhythm developed to efficiently deliver and consume innovations and improvements. In the case of Apple, the extra time was needed to prepare retailers, warehouses, and IT managers working in enterprises that used the software to prepare for the next release.

The nature of desktop software was causing longer lifecycles. The computer industry shipped a physical box containing manuals and floppy disks to the owner of the desktop computer. Advances in connecting desktop computers to the Internet had a profound effect on software development lifecycles. The *Internet Software Development Lifecycle* emerged to build browser-based software:

1. Specify the program from a mock-up of a Web site.
2. Write the software.
3. Unit test the application.
4. Fix the problems found in the unit test.
5. Internal employees test the application.
6. Fix the problems found.
7. Publish the software to the Internet.
8. Rapidly add minor bug fixes to the live servers.

In the above lifecycle, agile programming techniques swap steps 2 and 3. Agile programming recommends developing the test first, and then developing the code of the application.

The Internet spawned a new way of computing: Many little servers were connected together. Internet software is developed the same way, with many little modules connected together. Businesses hosting Internet applications are more comfortable introducing many minor bug fixes in succession than desktop system managers.

With so many desktop and server computers connected to the Internet, software development costs lowered to a level at which individual developers are able to write, publish, and support software. These developers are now experimenting with new ways to write software, including the following:

- **Open source development and distribution**. The software comes with the source code. Users finding bugs can either fix the problem themselves or participate in a community of developers to make the bugs known and fix them. If an open source project is vital, the changes needed by the user are accomplished.
- **Extreme (Agile or XP) programming**. Nodes of developers implement complex, large systems, each working on a different facet of the overall software. A contract is made between individual developers and the team to deliver needed software modules with a predefined and mutually agreed-upon API.

Starting in the late 1990s, the computer industry witnessed a stampede of software development projects. Supported by new software development tools, new software development lifecycles, and new ways to write software, the one area of development that remains firmly stuck in the 1990s are software testing tools, techniques, and methods.

Testing Methods

Software testing is as new as the computer industry itself—less than 40 years old. So it should be no surprise that new software testing methods appear every year. In the age of Internet software development, modern testing methods fall into these categories:

- Click-stream testing
- Unit testing (state, boundary, error, privilege)
- Functional system testing (transactions, end-to-end tests)

- Scalability and performance testing (load tests)
- Quality of Service Testing

Each of these techniques fills a void in a segment of the software development space. They all try to address every software developer's goal to deliver highly usable, productive, and quality applications. The following sections delve into definitions for these testing methods.

Click-Stream Testing

In 1996, Excite, a popular Web search engine company of its day, won praise from computer magazines by adding a personalization feature: Excite users could log in and set their preference to see weather information for their hometowns. This was a first-ever design. Soon after, the other search engine Web sites let you personalize features. This opened the age of the portal, or gateway, a Web site that enables users to personalize the site and to access deeper levels of content than is available to the general public. Prior to Excite's innovation, Web pages were updated manually by Web site editors—the Web functioned more like network television, where viewers changed channels to view single shows. Today, everyone expects Web-enabled applications to offer personalization, and Web sites that do not operate like desktop applications appear to be inconsequential.

In the early days, testing Web sites meant checking that the personalization worked and then checking the infrastructure to make sure the site continued to work. *Infrastructure* includes everything that exists between the user's browser and the database driving the Web-enabled application, and it includes network routers, switches, load balancers, server hardware, Web server software, and Common Gateway Interface (CGI) programs. CGIs were the first popular definition for software that runs on a Web server to provide dynamic content to a user.

Early Web-enabled application developers were interested in click-stream testing statistics. In a click-stream, the user clicks to move from one page to another on a Web site. The more clicks, the more money a Web site publisher earned in advertising or sponsorship of pages. The click-stream tools showed which URLs the users clicked, the Web site's user activity by time period during the day, and other data otherwise found in the Web server logs. Popular choices for click-stream testing statistics include KeyNote Systems Internet weather report, WebTrends log analysis utility, and the NetMechanic monitoring service.

Unfortunately, click-stream testing statistics reveal almost nothing about the user's ability to achieve their goals using the Web site. For example, a Web site may show a million page views, but 33% of the page views may simply be pages with the message "Found no search results." With click-stream testing, there's no way to tell when users reach their goals.

Unit Testing

Today, a single developer's personal choice of development tool is becoming less important as team-based software development using object-oriented technology has become the norm. Beginning in the 1990s, computer programming languages and operating system platforms have an object-oriented programming strategy. Consequently, the languages and platforms used to create these platforms look very similar to developers. Teams of developers can now work on software projects, with each developer using his or her own tools to produce object-oriented code.

In a team, individual engineers work on code modules and guarantee the modules' function. The team is bound together through a series of verbal and email contracts that define each module's functions and interfaces. The internal workings of each module are the responsibility of the software engineer. It is common sense, then, to expect that the engineer's primary concern will be making certain that a module is providing valid data when called, but the engineer doesn't care how the module is constructed internally. The engineer's job in ensuring that the module's responses are valid is called unit testing.

Unit testing finds problems and errors at the module level before the software leaves development. Unit testing is accomplished by adding a small amount of the code to the module that validates the module's responses. For example, a module may check the temperature of an air conditioner. Such a module written in Java may look like this:

```
class checktemperature {
  private int temperature = 0;

  public int gettemp()
  {
    return temperature;
  }

  public boolean validate( int temperature )
  {
    if ( temperature > 40 && temperature < 110 )
```

```
            return true;
       else
            return false;
     }
}
```

The checktemperature object has a method (gettemp) that returns the temperature found in the air conditioner. The developer of checktemperature also includes the validate method. The air conditioner temperature sensor is able to operate only within a range of 40 to 110 degrees Fahrenheit. If the temperature is actually 20 degrees, the returned value is invalid.

Validation methods provide a means for each code module to check itself for valid data when being compiled into a Web-enabled application. This is accomplished by running a unit test utility that operates at compile time. In addition to seeing compiler errors for syntax and construction problems, the unit test will validate the newly compiled object. Here is the output of a make utility that performs unit testing as it compiles the software modules:

```
Compiling bigproject.
Compiling checktemperature.
Running unit test on checktemperature, input value is 55.
Unit test successful.
Building bigproject file.
```

While unit testing is a huge win for developers, some developers can be lulled into a false sense of security. For example, a developer may think: "It passed the unit tests, so the software quality should be high." Unfortunately, however, unit tests can be meaningless if they're used as the sole measure to determine the health of a Web-enabled application.

Testing a Web-enabled application must also include intelligent end-to-end system tests. A system test checks the whole Web-enabled application, from the user operating a keyboard and mouse, to the Web-enabled application logic, to the database and communication systems underneath. If everything is not working, the user will not be able to accomplish his or her goals.

Functional System Testing

While unit testing is a fine technique used by developers, users will never operate only an individual module. Instead, users access collections of modules that make up the overall system. System tests check that the software

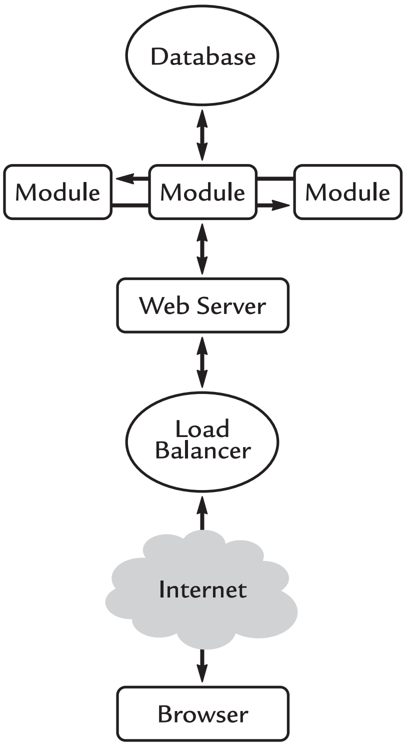


Figure 1–7 Components of a Web-enabled application.

functions properly from end-to-end. Figure 1–7 shows the components found in a production Web-enabled application data center.

Functional system tests check the entire application, from the client, which is depicted in Figure 1–7 as a Web browser but could be any application that speaks an open protocol over a network connection, to the database and everything in between. Web-enabled application frameworks deploy Web browser software, TCP/IP networking routers, bridges and switches, load-balancing routers, Web servers, Web-enabled application software modules, and a database. Additional systems may be deployed to provide directory service, media servers to stream audio and video, and messaging services for email.

A common mistake of test professionals is to believe that they are conducting system tests while they are actually testing a single component of the system. For example, checking that the Web server returns a page is not a system test if the page contains only a static HTML page. Instead, such a test checks the Web server only—not all the components of the system.

Scalability and Performance Testing

Scalability and performance testing is the way to understand how the system will handle the load caused by many concurrent users. In a Web environment concurrent use is measured as simply the number of users making requests at the same time. One of the central points of this book is that the work to perform a functional system test can and should be leveraged to conduct a scalability and performance test. The test tool you choose should be able to take the functional system test and run it multiple times and concurrently to put load on the server. This approach means the server will see load from the tests that is closer to the real production environment than ever before.

Quality of Service Testing

Understanding the system's ability to handle load from users is key to provisioning a datacenter correctly, however, scalability and performance testing does not show how the actual datacenter performs while in production. The same functional system test from earlier in this chapter can, and should, be reused to monitor a Web-enabled application. By running the functional system test over long periods of time, the resulting logs are your proof of the quality of service (QoS) delivered to your users. (They also make a good basis for a recommendation of a raise when service levels stay high.)

This section showed definitions for the major types of testing. This section also started making the case for developers, QA technicians, and IT managers to leverage each other's work when testing a system for functionality, scalability, and performance.

Next we will see how the typical behavior of a user may be modeled into an intelligent test agent. Test agents are key to automating unit tests, functional system tests, scalability and performance tests, and quality-of-service tests. The following sections delve into definitions for these testing methods.

Defining Test Agents

In my experience, functional system tests and quality of service tests are the most difficult of all tests, because they require that the test know something of the user's goals. Translating user goals into test agent code can be challenging.

A Web-enabled application increases in value as the software enables a user to achieve important goals, which will be different for each user. While it may be possible to identify groups of users by their goals, understanding a single user's goals and determining how well the Web-enabled application helped the user achieve those goals is the best way to build a test. Such a test will better determine a Web-enabled application's ability to perform and to scale than a general test. This technique also helps the test professional translate user goals into test agent code.

The formal way to perform system tests is to define a test agent that models an individual user's operation of the Web-enabled application to achieve particular goals. A test agent is composed of a checklist, a test process, and a reporting method, as described in Table 1–1.

Table 1–1 Components of an Intelligent Test Agent

Component	Description
Checklist	Defines conditions and states
Test process	Defines transactions needed to perform the checklist
Reporting method	Records results after the process and checklist are completed

Suppose a Web-enabled application provides travel agents with an online order-entry service to order travel brochures from a tour operator. The order-entry service adds new brochures every spring and removes the prior season's brochures. A test agent for the order-entry service simulates a travel agent ordering a current brochure and an outdated brochure. The test agent's job is to guarantee that the operation either succeeds or fails.

The following method will implement the example travel agent test by identifying the checklist, the test process, and the reporting method. The checklist defines the conditions and states the Web-enabled application will achieve. For example, the checklist for a shopping basket application to order travel brochures might look like this:

1. View list of current brochures. How many brochures appear?
2. Order a current brochure. Does the service provide a confirmation number?
3. Order an out-of-date brochure. Does the service indicate an error?

Checklists determine the desired Web-enabled application state. For example, while testing the Web-enabled application to order a current brochure, the Web-enabled application state is set to hold the current brochure; otherwise, the application is in an error state when an out-of-date brochure order appears.

A test agent process defines the steps needed to initialize the Web-enabled application and then to run the Web-enabled application through its paces, including going through the checklist. The test agent process deals with transactions. In the travel agent brochure order-entry system, the test agent needs these transactions:

1. Initialize the order entry service.
2. Look up a brochure number.
3. Order a brochure.

The transactions require a number of individual steps. For example, transaction 2 requires that the test agent sign in to the order-entry service, post a request to show the desired brochure number, confirm that the brochure exists, post a request to order the brochure, and then sign out.

Finally, a test agent must include a reporting method that defines where and in what format the results of the process will be saved. The brochure order-entry system test agent reports the number of brochures successfully ordered and the outdated brochures ordered.

Test agents can be represented in a number of forms. A test agent may be defined on paper and run by a person. Or a test agent may be a program that drives a Web-enabled application. The test agent must define a repeatable means to have a Web-enabled application produce a result. The more automated a test agent becomes, the better position a development manager will be in to certify that a Web-enabled application is ready for users.

The test agent definition delivers these benefits:

- Regression tests become easier. For each new update or maintenance change to the Web-enabled application software, the test agent shows which functions still work and which functions fail.
- Regression tests also indicate how close a Web-enabled application is ready to be accessed by users. The less regression, the faster the development pace.

- Developing test agents also provides a faster path to scalability and performance testing. Since a test agent models an individual user's use of a Web-enabled application, running multiple copies of the same test agent concurrently makes scalability and performance testing much more simple.

Scalability and Performance Testing with Test Agents

Testing Web-enabled applications is different than testing desktop software. At any time, a medium-scale Web-enabled application handles 1 to 5,000 concurrent users. Learning the scalability and performance characteristics of a Web-enabled application under the load of hundreds of users is important to manage software development projects, to build sufficient data centers, and to guarantee a good user experience. The interoperating modules of a Web-enabled application often do not show their true nature until they're loaded with user activity.

You can analyze a Web-enabled application in two ways: by scalability and performance. I have found that analyzing one without the other will often result in meaningless answers. What good is it to learn of a Web-enabled application's ability to serve 5,000 users quickly if 500 of those users receive error pages?

Scalability describes a Web-enabled application's ability to serve users under varying levels of load. To measure scalability, run a test agent and measure its time. Then run the same test agent with 1, 50, 500, and 5,000 concurrent users. Scalability is the function of the measurements. Scalability measures a Web-enabled application's ability to complete a test agent under conditions of load. Experience shows that a test agent should test 10 data points to deliver meaningful results, but the number of tests ultimately depends on the cost of running the test agent. Summarizing the measurements enables a development manager to predict the Web-enabled application's ability to serve users under load conditions.

Table 1–2 shows example scalability results from a test Web-enabled application. The top line shows the results of running a test agent by one user. In the table, 85% of the time the Web-enabled application completed the test agent in less than one second; 10% of the time the test agent completed in less than 6 seconds; and 5% of the time the test agent took 6 or more seconds to finish.

Table 1–2 Example Results Showing Scalability of a Web Service

	<1 second	2–5 seconds	>5 seconds
1	85%	10%	5%
50	75%	15%	10%
500	70%	20%	10%
5000	60%	25%	15%

Notice in the table what happens when the Web-enabled application is put under load. When 50 users concurrently run the same test agent, the Web-enabled application does not perform as well as it does with a single user. With 50 users, the same test agent is completed in less than 1 second only 75% of the time. The Web-enabled application begins to suffer when 5,000 users begin running the test agent. At 5,000 users, only 60% will complete the test agent in less than 1 second.

One can extrapolate the scalability results for a Web-enabled application after a minimum of data points exists. If the scalability results contained tests at only 1 and 50 users, for example, the scalability extrapolation for 5,000 users would be meaningless. Running the scalability tests with at least four levels of load, however, provides meaningful data points from which extrapolations will be valid.

Next we look at performance indexes. Performance is the other side of the coin of scalability. Scalability measures a Web-enabled application's ability to serve users under conditions of increasing load, and testing assumes that valid test agents all completed correctly. Scalability can be blind to the user experience. On the other hand, performance testing measures failures.

Performance testing evaluates a Web-enabled application's ability to deliver functions accurately. A performance test agent looks at the results of a test agent to determine whether the Web-enabled application produced an exceptional result. For example, in the scalability test example shown in Table 1–3, a performance test shows the count of error pages returned under the various conditions of load.

Table 1–3 shows the performance results of the sample example Web-enabled application whose scalability was profiled in Table 1–2. The performance results show a different picture of the same Web-enabled application. At the 500 and 5,000 concurrent-user levels, a development manager looking

Table 1–3 Example Performance Test Agent Results

Performance	<1 second	2–5 seconds	>6 seconds	Total
1	1%	5%	7%	13%
50	2%	4%	10%	16%
500	4%	9%	14%	27%
5000	15%	25%	40%	80%

solely at the scalability results might still decide to release a Web-enabled application to users, even though Table 1–2 showed that at 500 concurrent users 10% of the pages delivered had very slow response times—slow test agents in this case are considered to take 6 or more seconds to complete. Would the development manager still release the Web-enabled application after looking at the performance test results?

Table 1–3 shows the Web-enabled application failed the test 15% of the time when the test agent completed the test in less than 1 second while serving at the 5,000 concurrent user level. Add the 25% value for test agents that complete in 5 seconds or less and the 40% for test agents that complete in 6 or more seconds, and the development manager has a good basis for expecting that 80% of the users will encounter errors when 5,000 users concurrently load the Web-enabled application.

Both scalability and performance measures are needed to determine how well a Web-enabled application will serve users in production environments. Taken individually, the results of these two tests may not show the true nature of the Web-enabled application. Or even worse, they may show misleading results!

Taken together, scalability and performance testing shows the true nature of a Web-enabled application.

Testing for the Single User

Many developers think testing is not complete until the tests cover a general cross-section of the user community. Other developers believe high-quality software is tested against the original design goals of a Web-enabled application as defined by a product manager, project marketer, or lead developer.

These approaches are all insufficient, however, because they test toward the middle only.

Testing toward the middle makes large assumptions of how the aggregate group of users will use the Web-enabled application and the steps they will take as a group to accomplish common goals. But Web-enabled applications simply are not used this way. In reality, each user has their own personal goal and method for using a Web-enabled application.

Intuit, publishers of the popular Quicken personal finance management software, recognized the distinctiveness of each user's experience early on. Intuit developed the "Follow me home" software testing method. Intuit developers and product managers visited local software retail stores, waiting in the aisles near the Intuit products and watching for a customer to pick up a copy of Quicken. When the customer appeared ready to buy Quicken, the Intuit managers introduced themselves and asked for permission to follow the customer home to learn the user's experience installing and using the Quicken software.

Intuit testers could have stayed in their offices and made grand speculations about the general types of Quicken users. Instead, they developed user archetypes—prototypical Web-enabled application users based on the real people they met and the experience these users had. The same power can be applied to developing test agents. Using archetypes to describe a user is more efficient and more accurate than making broad generalizations about the nature of a Web-enabled application's users. Archetypes make it easier to develop test agents modeled after each user's individual goals and methods of using a Web-enabled application.

The best way to build an archetype test agent is to start with a single user. Choose just one user, watch the user in front of the Web-enabled application, and learn what steps the user expects to use. Then take this information and model the archetype against the single user. The better an individual user's needs are understood, the more valuable your archetype will be.

Some developers have taken the archetypal user method to heart. They name their archetypes and describe their background and habits. They give depth to the archetype so the rest of the development team can better understand the test agent.

For example, consider the archetypal users defined for Inclusion Technologies, one of the companies I founded, Web-enabled application software. In 1997, Inclusion developed a Web-enabled application to provide collaborative messaging services to geographically disbursed teams in global corporations. Companies like BP, the combined British Petroleum and Amoco energy companies, used the Inclusion Web-enabled application to build a secure private extranet, where BP employees and contractors in the financial auditing groups could exchange ideas and best practices while performing their normal work.

Test agents for the BP extranet were designed around these archetypal users:

- Jack, field auditor, 22 years old, recently joined BP from Northwestern University, unmarried but has a steady girlfriend, has been using spreadsheet software since high school, open to using new technology if it gets his job done faster, loves motocross and snow skiing.
- Madeline, central office manager, 42 years old, married 15 years with two children, came up through the ranks at BP, worked in IT group for three years before moving into management, respects established process but will work the system to bring in technology that improves team productivity.
- Lorette, IT support, 27 years old, wears two pagers and one mobile phone, works long hours maintaining systems, does system training for new employees, loves to go on training seminars in exotic locations.

The test agents that modeled Jack's goals concentrate on accessing and manipulating data. Jack often needs to find previously stored spreadsheets. In this case, a test agent signs in to the Web-enabled application and uses the search functions to locate a document. The test agent modifies the document and checks to make sure the modifications are stored correctly.

The test agent developed for Madeline concentrates on usage data. The first test agent signs in to the Web-enabled application using Madeline's high-level security clearance. This gives permission to run usage reports to see which of her team members is making the most use of the Web-enabled application. That will be important to Madeline when performance reviews are needed. The test agent will also try to sign in as Jack and access the same reports. If the Web-enabled application performs correctly, only Madeline has access to the reports.

Test agents modeled after Lorette concentrate on accessing data. When Lorette is away from the office on a training seminar, he still needs access to the Web-enabled application as though she were in the office. The test agent uses a remote login capability to access the needed data.

Understanding the archetypes is your key to making the test agents intelligent. For example, a test agent for Lorette may behave more persistently than a test agent for Madeline. If a test agent tries to make a remote connection that fails, the test agent for Lorette would try again and then switch to a difference access number.

Creating Intelligent Test Agents

Developing test agents and archetypes is a fast, predictable way to build good test data. The better the data, the more you know about a Web-enabled application's scalability and performance under load. Analyzing the data shows scalability and performance indexes. Understanding scalability and performance shows the expenses a business will undertake to develop and operate a high-quality Web-enabled application.

In many respects, testing Web-enabled applications is similar to a doctor treating patients. For example, an oncologist for a cancer patient never indicates the number of days a patient has left to live. That's simply not the nature of oncology—the study and treatment of cancer. Oncology studies cancer in terms of epidemiology, whereby individual patient tests are meaningful only when considered in collection with a statistically sufficient number of other patients. If the doctor determines an individual patient falls within a certain category of overall patients, the oncologist will advise the patient what all the other patients in that category are facing. In the same way, you can't test a Web-enabled application without using the system, so there is no way to guarantee a system will behave one way or the other. Instead you can observe the results of a test agent and extrapolate the performance and scalability to the production system.

Accountants and business managers often cite the law of diminishing returns, where the effort to develop one more incremental addition to a project provides less and less returns against the cost of the addition. Sometimes this thinking creeps into software test projects.

You can ask yourself, at what point have enough test agents and archetypes been used to make the results meaningful? In reality, you can never use enough test agents and user archetypes to guarantee a Web-enabled application's health. While an individual test agent yields usable information, many test agents are needed. Each test agent needs to profile an individual user's goals and methods for using a Web-enabled application. Taken together, the test agents show patterns of scalability and performance.

Data from many test agents develops predictability. Each new Web-enabled application or maintenance release may be tested against the library of test agents in a predictable time and with a predictable amount of resources.

Automated Testing

The overhead of developing and running test agents can become too much of a burden when performed manually. After the first agent is developed, it is time for a second. There may be no end to this cycle. As a result, choosing to build test agents modeled around individual users requires an increasingly costly test effort—or it requires automation. Test automation enables your library of test agents to continue growing, and testing costs can remain manageable.

Test automation enables a system to run the test agents concurrently and in bulk. Many automated test agents will drive a Web-enabled application at levels reaching normal for the real production environment. The amount of users depends on your expected Web-enabled application users. Experience tells us to multiply the expected number of users by 10, and test against the result.

Each automated test agent embodies the behavior of an archetype. Multiple concurrent-running copies of the same test agent will produce interesting and useful test results. In real life, many users will exhibit the same behavior but at different times. Intelligent test agents bring us much closer to testing in a real-world production environment.

Summary

In this chapter, we look at the current computing industry state. We found that software developers, QA technicians, and IT managers have the largest selection of tools, techniques, hardware, and knowledge to build integrated Web-enabled software applications ever. We also found that choosing tools, techniques, and methods impacts system scalability and reliability.

We learned that building high-quality Web-enabled applications requires tools, methodologies, and a good understanding of the development team's behavior. In Chapter 2, we will see in detail how intelligent test agents provide useful and meaningful data and how to determine how close the user gets to meeting his or her needs while using the Web-enabled application. We will also see how intelligent test agents automate running of test suites and discuss the test environments and test tools necessary to understand the test suite results.

This chapter shows the forces at work that have made Web-enabled applications so popular and shows how you can achieve entirely new levels of productivity by using new software testing tools, new techniques, and new methods.

2 When Application Performance Becomes a Problem

Software development professionals have traditionally quantified good distributed system performance by using well-established quality methods and criteria. This chapter shows how a new breed of methods and criteria are needed to test Web-enabled applications. The new software test techniques help developers, QA technicians, and IT managers prioritize their efforts to maximize the user's ability to achieve their personal goals. New tools introduced in this chapter, for example, the Web Rubric and Web Application Points System, provide easy-to-follow criteria to determine the health of a Web-enabled application.

Just What Are Criteria?

By now in the Internet revolution, it is common sense that Internet users expect Web-enabled applications to perform at satisfactory levels, and therein lies the dilemma: one user's satisfaction is another's definition of frustration. In addition, a single user's satisfaction criteria changes over time. Figure 2–1 shows how user criteria can change over time.

Figure 2–1 shows that although it may be fine to launch a Web-enabled application according to a set of goals deemed satisfactory today, the goals will change over time. In the early days of the Web, simply receiving a Web page satisfied users. Later, users were satisfied only with fast-performing Web sites. Recently, users commonly make choices of Web-enabled applica-

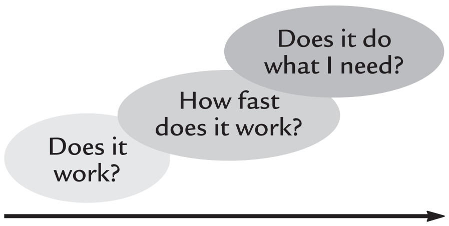


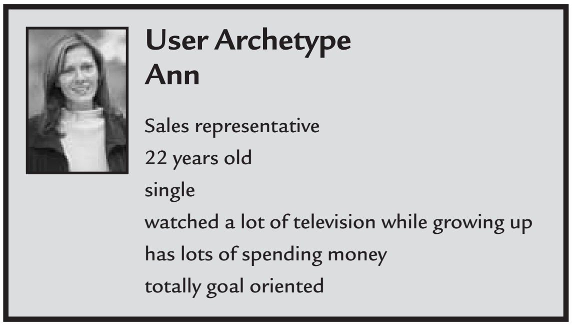
Figure 2–1 A user's changing needs.

tions based on the service's ability to meet their personal needs immediately. The pages that appear must have meaningful and useful data that is well organized. And, of course, it must appear quickly.

With user expectations changing over time, the tests of a system conducted on one day will show different results on a second day. In my experience, Web-enabled applications need to be constantly tested, monitored, and watched. In a retail store-front analogy, the Web-enabled application store is open 24 hours a day with 1 to 100,000 customers in the store at any given time. Bring in the intelligent test agents!

Intelligent test agents, introduced in Chapter 1, are a fast and efficient way to test a Web-enabled application everyday. At their heart, intelligent test agents automate the test process so that testing is possible everyday. Building agents based on user archetypes produces meaningful data for determining a Web-enabled application's ability to meet user needs. As user satisfaction goals change, we can add new test agents based on new user archetypes.

To better understand user archetypes, I will present an example. Figure 2–2 shows a typical user archetype description.



User Archetype
Ann

Sales representative
22 years old
single
watched a lot of television while growing up
has lots of spending money
totally goal oriented

Figure 2–2 An example user archetype: Ann is a sales representative for a manufacturing business. The more personable the user archetype definition, the more your team will respond!

Defining user archetypes in this way is up to your imagination. The more time you spend defining the archetype, the easier it will be for all your team members to understand. Using this archetype technique will achieve these results in your team:

- Archetypes make an emotional connection between the goals of a prototypical user of the application and the software developer, QA technician, and IT manager that will deliver the application. Your team members will have an easy-to-understand example of a person that prototypically will use your Web-enabled application. Rather than discussing individual features in the application, the team will be able to refer to functions that the archetype will want to regularly use to accomplish their personal goals.
- Archetypes bring discussions of features, form, and function in your Web-enabled application from a vague general topic to a focused user goal-oriented discussion. For example, instead of discussing the user registration page's individual functions, the team discussion of the user registration page will cover how Ann uses the registration page.
- Archetypes give the team an understanding of where new functions, bugs, and problems need to be prioritized in the overall schedule. That is because Ann needs solutions rather than some unknown user.

What's more, user archetypes make it much easier to know what to test and why. For example, a test agent modeled after Ann will focus on the speed at which she accomplishes her goals. A test agent might even go so far as to drive a Web-enabled application to deliver a search result and then cancel the search if it takes longer than 2 seconds to complete—that is because Ann is goal oriented with a short attention span. The test agent modeled after Ann is added to the library of other test agents, including test agents for Jack, Madeline, and Lorette that were introduced in Chapter 1.

Ultimately, developing new user archetypes and test agents modeled after the archetype's behavior produces automated Web tests that get close to the real experience of a system serving real users. What's more, user archetypes and test agents allow us to know when Web-enabled application performance becomes a problem.

Defining Criteria for Good Web Performance

Before the Internet, client/server network systems operated on private networks and were accessed at rather predictable times, with simple patterns, by a well-known and predictable group of people. Many times, the network usually ran in a single office with maybe a few remote locations.

Advances in routers, gateways, bridges, and other network equipment introduced the era where any device on the office Local Area Network (LAN) may receive requests from people anywhere in the world and at any time. An office network may not be exposed to the open Internet, but it is likely that employees will access the office network from home, remote offices will access the network through a network bridge, and the LAN may also be handling traffic for your phone system using Voice over Internet Protocol (VoIP). Your Web-enabled applications on the LAN are subjected to highly unpredictable load patterns created by a widely heterogeneous and unpredictable group of users.

So what is there to lose if your company chooses to ignore load patterns? Few organizations reward uptime achievements for Web-enabled application infrastructure. It's the downtime, stupid! Excessive loads cause serious harm to a company's bottom line, market value, and brand equity—not to mention the network manager's reputation. For example, you might recall when eBay went down for 22 hours in 1999 due to a load-related database error. The company lost $2 million in revenues and eBay stock lost 10 percent of its value as a result (details are at http://www.internetnews.com/ec-news/article.php/4_137251). Although most businesses are not as large as eBay, they will suffer proportionally should load patterns be ignored.

Paying attention to load patterns and choosing the appropriate test methodology is critical to the validity of such testing. Dependable and robust test methodology for Web-enabled applications deployed on the Internet or in extranets is mandatory today. With poor methodology, the results are at best useless, and in the worst case, misleading.

Defining criteria for good Web-enabled application performance has changed over the years. With so much information available, it's relatively easy to use current knowledge to identify and update outdated criteria. Common "old" testing techniques include ping tests, click-stream measurement tools and services, and HTML content checking.

- **Ping tests** use the Internet Control Message Protocol (ICMP) to send a ping request to a server. If the ping returns, the server is assumed to be alive and well. The downside is that usually a Web server will continue to return ping requests even when the Web-enabled application has crashed.
- **Click-stream measurement tests** makes a request for a set of Web pages and records statistics about the response, including total page views per hour, total hits per week, total user sessions per week, and derivatives of these numbers. The downside is that if your Web-enabled application takes twice as many pages as it should for a user to complete his or her goal, the click-stream test makes it look as though your Web site is popular, while to the user your Web site is frustrating.
- **HTML content-checking tests** makes a request to a Web page, parses the response for HTTP hyperlinks, requests hyperlinks from their associated host, and if the links returned successful or exceptional conditions. The downside is that the hyperlinks in a Web-enabled application are dynamic and can change, depending on the user's actions. There is little way to know the context of the hyperlinks in a Web-enabled application. Just checking the links' validity is meaningless, if not misleading.

Understanding and developing the criteria for good Web-enabled application performance is based on the users' everyday experiences. Companies with Web-enabled applications that have relied solely on ping tests, click-stream measurements, or HTML content develop useless and sometimes misleading results. These tools were meant to test static Web sites, not Web-enabled applications.

In a world where users choose between several competing Web-enabled applications, your choice of criteria for good Web-enabled application performance should be based on three key questions:

- Are the features working?
- Is performance acceptable?
- How often does it fail?

Are the Features Working?

Assemble a short list of basic features. Often this list may be taken from a feature requirements document that was used by the software developers that created the Web-enabled application. While writing down the features list, consider the user who just arrived for the first time at your Web-enabled application and is ready to use it.

Here's an example showing the basic features list for the online support service at my company PushToTest:

1. Sign in and sign out.
2. Navigate to a discussion message.
3. Download a document.
4. Post a message.
5. Search for a message using key words.

While the PushToTest Web site has more than 480 server objects, 13,000 lines of code, and a versatile user interface, it comes down to the five features listed above to guarantee that the application was working at all.

Is Performance Acceptable?

Check with three to five users to determine how long they will wait for your Web-enabled application to perform one of the features before they abandon the feature and move on to another. Take some time and watch the user directly, and time the seconds it takes to perform a basic feature.

How Often Does It Fail?

Web-enabled application logs, if formatted with valuable data, can show the time between failures. At first, developing a percentage of failures acceptable to users may be appetizing. In reality, however, such a percentage is meaningless. The time between failures is an absolute number. Better to estimate the acceptable number first, and then look into the real logs for a real answer. The Web rubric described in the next section is a good method to help you understand failure factors and statistics.

Web-Enabled Application Measurement Tools

In my experience, the best measurements for a Web-enabled application include the following:

1. Meantime between failures in seconds
2. Amount of time in seconds for each user session, sometimes known as a transaction
3. Application availability and peak usage periods
4. Which media elements are most used (for example, HTML vs. Flash, JavaScript vs. HTML forms, Real vs. Windows Media Player vs. QuickTime)

Developing criteria for good Web-enabled application performance can be an esoteric and difficult task. At this time, a more down-to-earth example of a method to define good performance criteria is in order.

The Web Rubric

In developing criteria for Web-enabled application performance, testing methods often include too much subjectivity. Many times, a criteria assessment grades a Web-enabled application well, but then when the grade is examined, the assessment criteria is vague and the performance behavior is overly subjective. That puts every tester into an oddly defensive position trying to justify the test results. A Web rubric is an authentic assessment tool that is particularly useful in assessing Web performance criteria where the Web-enabled application results are complex and subjective.

Authentic assessment is a scientific term that might be better stated as "based in reality." Years ago, apprenticeship systems assessed people based on performance. In authentic assessment, an instructor looks at a student in the process of working on something real, provides feedback, looks at the student's use of the feedback, and adjusts the instruction and evaluation accordingly.

A Web rubric is designed to simulate real-life activity to accommodate an authentic assessment tool. It is a formative type of assessment because it becomes a part of the software development lifecycle. It also includes the developers themselves, who assess how the Web-enabled application is performing for users. Over time, the developers can assist in designing subsequent versions of the Web rubric. Authentic assessment blurs the lines between developer, tester, and user.

Table 2–1 is an example of a Web rubric for email collaboration Web service.

Table 2–1 A Rubric for an Email-Enabled Web Application

Criteria assessed through system use	Level 1 Beginning	Level 2 Developing	Level 3 Standard	Level 5 Above standard
Basic features are functioning	Few features work correctly the first time used.	Many features do not operate. Some missing features required to complete the work.	Most features operate. Workarounds available to complete work.	All features work correctly every time they are used.
Speed of operation	Many features never completed.	Most features completed before user lost interest.	Most features completed in 3 seconds or less.	All features complete in less than 3 seconds.
Correct operation	Few features result successfully with an error condition.	Some features end in an error condition.	Most features complete successfully.	All features complete successfully.

The approval criteria from this rubric is as follows:
The highest, most consistent score on the criteria must be at level 3, with no score at level 1.
For Web-enabled applications with three criteria, two must be at a level 3. The remaining criteria must be higher than level 1.

The advantages of using rubrics in assessment are as follows:

- Assessment is more objective and consistent.
- The rubric can help the tester focus on clarifying criteria into specific terms.
- The rubric clearly describes to the developer how his or her work will be evaluated and what is expected.
- The rubric provides benchmarks against which the developer can measure and document progress.

Rubrics can be created in a variety of forms and levels of complexity; however, they all contain common features that perform the following functions:

- Focus on measuring a stated objective (performance, behavior, or quality)
- Use a range to rate performance
- Contain specific performance characteristics arranged in levels indicating the degree to which a standard has been met

While the Web rubric does a good job of removing subjectivity from good Web-enabled application performance criteria, good Web-enabled application performance can be defined in other important ways.

The Four Tests of Good Performance

Web-enabled application performance can be measured in four areas:

- **Concurrency.** A measurement taken when more than one user operates a Web-enabled application. You can say a Web-enabled application's concurrency is good when the Web-enabled application can handle large numbers of concurrent users using functions and making requests. Employing load-balancing equipment often solves concurrency problems.
- **Latency.** A measurement of the time it takes a Web-enabled application to finish processing a request. Latency comes in two forms: the latency of the Internet network to move the bits from a browser to server, and software latency of the Web-enabled application to finish processing the request.
- **Availability.** A measurement of the time a Web-enabled application is available to take a request. Many "high availability" computer industry software publishers and hardware manufacturers will claim 99.9999% availability. As an example of availability, imagine a Web-enabled application running on a server that requires 2 hours of downtime for maintenance each week. The formula to calculate availability is (Total hours – downtime hours) / total hours. As each week consists of 168 total hours (7 days times 24 hours per day), a weekly 2-hour downtime results in 98.8095% availability [(168 – 2) / 168].
- **Performance.** This is a simple average of the amount of time that passes between failures. For example, an application that

threw errors at 10:30 AM, 11:00 AM, and 11:30 AM has a performance measurement of 30 minutes.

Over the years I found that many times these terms are confused and used interchangeably. I am not immune from such foibles too! A handy technique to remember these four tests of good performance is to think of someone CLAP-ing their hands. CLAP is a handy acronym for concurrency, latency, availability, and performance that helps me remember the test area's names.

Components of a Good Test Agent

The Web rubric provides a systematic way of testing a Web-enabled application. Complexity and subjectivity are avoided by using a Web rubric to define the testing criteria. With a Web rubric in hand, special care must be taken to implement criteria correctly. Subjectivity and laxness can creep into a test.

As the old expression goes: *Garbage in, garbage out.*

The modern expression for testing Web-enabled applications might go like this: *Who's watching for garbage?*

That is because there still is not much of a defined profession for software testing. Few people in the world are software test professionals. And few text professionals expect to remain in software testing throughout their careers. Many software test professionals I have known view their jobs as steppingstones into software development and management.

When developing criteria for Web-enabled application performance, the testers and the test software need to show many unique qualities, including a nature to be rigorous, systematic, and repeatable. These traits do not guarantee success. However, recognizing the traits of good software test people and good software test agents can be an important factor to ensure that the Web-enabled application test criteria are tested correctly. Table 2–2 shows the major traits I look for in a software test person and how the same traits are found in intelligent test agents.

Recognizing the traits of good software test people and good software test agents are important to ensuring the Web service test criteria is tested correctly.

Table 2–2 Traits to Look for in Test Agents and Test People

Intelligent software test agents	Software test people
Rigorous—intelligent software test agents are "multipathed." While driving a Web-enabled application, if one path stops in an error condition, a rigorous test agent will try a second or third path before giving up.	**Rigorous**—test people's nature drives them to try to make a broken piece of software work in any way they can. They are multipathed and creative.
Systematic—test agents run autonomously. Once installed and instructed on what to do, they do not need reminding to follow test criteria.	**Systematic**—test people always start and end with the same process while following a test criteria; however, they concentrate on finding problems the system was not defined to find.
Repeatable—test agents follow detailed instructions describing the state a system must be in to run a test criteria.	**Repeatable**—test people find something new each time they run a test criteria, even though the test is exactly the same each time.

Web-Enabled Application Types

In the early days of the Internet and the Web, it appeared that Web-enabled applications would just be another form of software development. It should be apparent today, however, that all software will include a Web connectivity strategy. Even PC video games include features to receive automatic software updates and allow players to play against users across a network connection.

No longer does a software category of Web-enabled applications exist, since all software includes Web-enabled application qualities! This fact has had a profound effect on developing criteria for software performance. For example, no single criteria exists for connection speed performance. The Web-enabled application that powers a TiVo video hard-drive recorder appliance connects to a central server every day early in the morning to download a program guide of TV shows. If its connection speed drops down to 50% of normal transfer speed but still completes the download process, who really cares? The connection speed performance criteria apply to that video hard-drive recorder only. No one has developed criteria that would work for both the video recorder and a PC video game, as both are in distinctly different categories of software, with their own performance criteria.

Does this mean that every new software program will require its own criterion? As major categories of Web-enabled application software emerge, each category will include resources to use for determining performance criteria and test tools to use for evaluating performance according to the criteria. I have lost count of the number of times when a technology industry leader announced convergence, only to find divergence and more datacenters filled with heterogeneous systems.

The major categories of Web-enabled application software today are:

- Desktop productivity software
- Utility software
- E-commerce software
- Network connectivity software
- Collaboration software
- Database software
- Directory and registry software
- Middleware
- Drivers, firmware, and utilities
- Storage software
- Graphic and human user interface software

Each of these types has its own Web-enabled application strategy and criterion for good performance.

When evaluating a new software category, look for tools that help determine the criteria for performance.

- As the system grows, so does the risk of problems such as performance degradation, broken functions, and unauthorized content. Determine the test tool's ability to continue functioning while the system grows.
- Individual tool features may overlap, but each type of tool is optimized to solve a different type of problem. Determine the best mix of test tools to cover the criteria.
- Diagnostic software locates broken functions, protecting users from error messages and malfunctioning code. Find a test tool that handles error conditions well.
- Look for a product that checks the integrity of Web-enabled applications. Integrity in a testing context means security and

functionality checks. For example, can you sign in and access unprivileged controls?

- A Web-enabled application's functions and content normally require separate test software. For example, although a button may function correctly, the button's label is misspelled. Additionally, the application publisher may be legally obliged to check the accessibility of the application by users with special needs. For example, U.S. government Section 508 rules are just one of many government regulations that require full accessibility for people with special needs.
- Complex systems manage user access to data and functions. A test tool needs to check authorization while conducting a Web performance criteria test.
- Performance monitoring tools need to measure Web-enabled application performance.
- A tool that tracks viewer behavior can help you optimize your Web-enabled application's content, structure, and functionality.

Eventually, major Web-enabled application categories emerge to become part of our common language. Web-enabled applications showing this kind of maturity are written to assume that no printed manual for the services will ever exist. Instead, the services' functions and usage will achieve a certain set of expectations of users in such a way that instruction will be unnecessary.

Software testing is a matter of modeling user expectations and ensuring that the software meets these expectations 24 hours a day, seven days a week. Developing criteria for good Web-enabled application performance is key to reaching this goal.

The Web-Enabled Application Points System (WAPS)

Testing Web-enabled applications can lead a development team in several different directions at once. For example, unit tests may show three out of ten software modules failing and response times lagging into the 10-plus second range, and the Web-enabled application may be returning system failure messages to users seven times in a day. A Web-enabled application that exhibits any of these problems is already a failure. Experience tells us that users find unacceptable Web-enabled applications that exhibit any of these characteristics.

At the same time, such results can pull development managers in many directions at once. With limited resources, which problem should take highest priority? The Web-enabled Applications Points System (WAPS, for short) is a good solution that helps development managers understand a Web-enabled application's health and prioritize a development team's efforts to solve problems and maintain the code. WAPS is an ideal measure when used in a rubric for good system performance.

WAPS is a quick, easy way to profile the performance and scalability of any Web-enabled application. By reducing the number of points, the team increases a Web-enabled application's ability to serve users well. The Points System combines three measurements, described in Table 2–3.

Table 2–3 The Web-Enabled Applications Points System

Criteria	Description	Points (reset every 24 hours)
Functions	Number of wrong states, boundary errors, and privilege problems.	1 point for every log entry showing functional errors.
Performance	Number of seconds to deliver dynamic Web pages.	2 points for every page that took more than 10 seconds to compile.
Failures	Number of failure occurrences in which a user sees an error message: e.g., "Sorry the information center encountered a database error."	3 points where the Web-enabled applications failed.

Count the number of points for each 24-hour period. Check the Web-enabled application's health from the categories shown in Table 2–4.

WAPS provides a metric to help developers understand the relative health of a software project—during initial development and then during maintenance. Effective communication to the development team is key to making WAPS an effective tool. You can automate the WAPS report so that developers receive a report every day. The less manual collection of WAPS data, the better.

Table 2–4 Points System Analysis

Points	Health	What you should do
1–10	Excellent	Enjoy the day and check the numbers again tomorrow.
11–25	Fair	Fix the problems that will get the count into the Excellent category; however, fix a mix of the functional, performance, and failure problems. Avoid concentrating on a single category of problems.
26–100	Poor	Your Web-enabled application is not ready for users. If your Web-enabled application is already in production, take it out of production, fix the problems, and explain these steps to the users.
101 or more	Horrid	Don't shoot yourself, there is a way to deliver healthy Web services. Assess your code writing skills, the project's complexity, and the major problem areas. Pick a date at least one week in the future and see if you have moved the Web service from the Horrid state to the Poor state. Then check the numbers again daily.

WAPS daily measurements already provide a lot of information to the developers; yet charting points over time can be invaluable to maintaining the health of a software project. PushToTest, a software test automation solutions company, uses WAPS and noted the data shown in Figure 2–3 over time for one of its Web-enabled applications.

WAPS results are presented at the start of the weekly developers' meeting. PushToTest engineers noted that WAPS places the focus on fixing problems

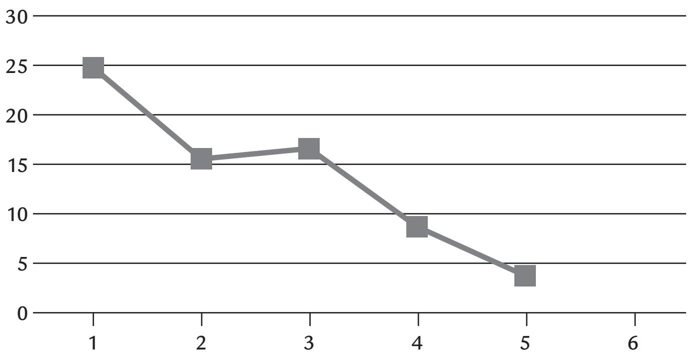


Figure 2–3 Mapping the WAPS system results for a site over time.

users encounter every day, and because WAPS is composed of multiple weighted types of problems, the developers could focus on targeted problems rather than just working down a list of problems sequentially.

So how does WAPS scale to large Web sites? Imagine applying WAPS to a public Internet site that features free music downloads. Ten million users may visit such a site every month. Would one still give a Web-enabled application a Horrid ranking if only 120 points accrued for an installed base of 10 million users? Yes! You may choose to weight the points table according to the size of the system being evaluated; however, counting points, as opposed to relying on percentages, is important.

Those users who ran into problems did not reach their personal goal because of the horrid nature of the Web-enabled application. WAPS treats every single user as critical to success. When a manager ignores the individual user and tries to play the numbers by fixing only problems that are significant to the aggregate target audience, that is usually the time that the one important magazine reviewer, venture capitalist, business development executive, or key customer tries to access the site and encounters a problem, and the entire development effort and company credentials suffer as a result.

WAPS should become part of the company language. Even outside the development team, the company will grow to know how the Web-enabled application is performing by looking at the daily WAPS report. Those workers who provide customer service and support, and even the marketing teams, will know when to apply more resources, maybe even more overtime, depending on the points the software racked up.

Another clear win for WAPS is the effect it has on developers. Web-enabled applications are complicated, intricate, and cumbersome systems to build, deploy, and manage. From the developers' perspective, building a Web-enabled application can be as intimidating as climbing a wall of ice—one slip and it's over. This causes many developers to be overly cautious and almost too focused on finishing. (And, of course, draconian managers could offer incentive pay based on WAPS!)

Developers direct their attention to these goals:

- **Design**—How the software modules interact
- **Features**—The code that performs a user-driven function
- **Performance**—The speed at which the software completes user-driven commands
- **Scalability**—How the software would act at different levels of usage

The obvious missing goal is usability. Who is responsible for the user's experience on the Web-enabled application?

The developers' best intentions are to deliver software that scales, performs well, and is usable. But when a software project goes over budget, behind schedule, and appears overly complex, many developers become conservative and defensive. Developers may even fall back on an often-cited and overused bromide: *Quality, features, and schedule. Choose any two of these.*

In this case, I recommend using WAPS. By agreeing to a scoring system, the development team has a tool to get back to a rational schedule and reasonable management expectations. No matter the size or complexity of the project, WAPS brings a measured balance to solving problems and delivering the finished Web-enabled application with usability intact.

The Web-Enabled Application's Framework

Web-enabled applications succeed when the system can scale from 1 user to 10,000 users without changing the software implementation or design. Web-enabled applications succeed when all users have good experiences, even though one user may use only a single function and other users use all the functions. Web-enabled applications succeed when the Web-enabled application is available whenever a user shows up. Testing Web-enabled applications under real-life stresses and situations make it possible to predict a Web-enabled application's success and to monitor the Web-enabled application to know when maintenance is needed. Choosing the appropriate components to a system that hosts a Web-enabled application greatly determines how successful a Web-enabled application will be.

With that in mind, this section presents a framework for building and deploying Web-enabled applications, instructions on how to apply intelligent test agents to Web-enabled applications, and a free open source set of tools and scripting language called TestMaker that can help build automated intelligent test agents. The result is a system and methodology that builds better software faster.

The Flapjacks Architecture

In my years as a designer and builder of Web systems, I've developed an analogy that aptly describes Web application architectures that consist of deploying many small servers that are accessed through a load balancer to provide a front end to a powerful database server. The analogy goes like this:

Hungry patrons (Web users) show up for breakfast at your (the developer's) diner. The more patrons that come in for breakfast, the more flapjacks (servers) you have to toss on the griddle. Some of the patrons want banana pancakes, some want blueberry, some want plain cakes with just a hint of vanilla, and many will want to sample several flavors. In the Web-enabled applications scenario, some users will require access to the servers running Java servlets, some will need application servers, some will need .NET and SOAP Web-enabled applications, and others will be looking for directory servers, but most will favor a combination.

All the flapjacks are dished out from the same batter. In this analogy, the batter functions as a single database for persistent storage, search, indexing, and so on.

The waitress takes orders (user requests) and passes them to the appropriate cook (say, the blueberry pancake chef), much like a load balancer routes Web browsers to the appropriate server (or an alternative, if necessary).

Figure 2–4 illustrates the "flapjacks architecture."

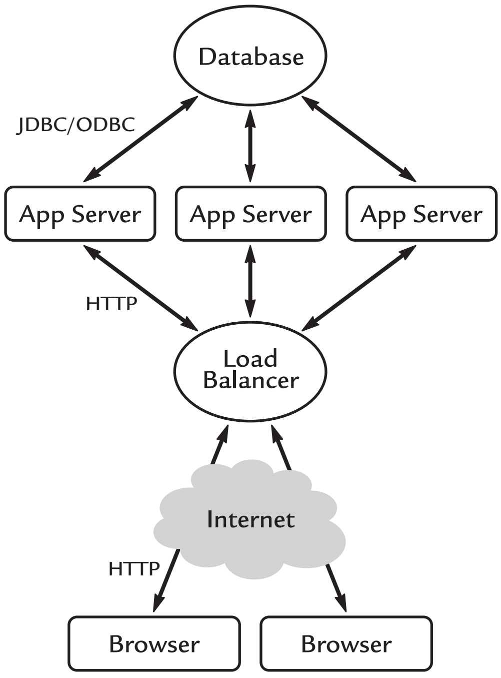


Figure 2–4 The flapjacks architecture.

This type of architecture is popular. The systems providing services on eBay, Yahoo!, HotMail, and other high-volume public Web sites are based on the flapjacks architecture—sometimes known as N-Tier architecture. Enterprise applications software providers, including the Microsoft .NET Framework, IBM WebSphere, and BEA WebLogic, recommend using the flapjacks architecture.

The flapjacks architecture has many benefits to offer Web-enabled applications, especially for private intranet systems. Users tend to get faster performance and less waiting times from small, inexpensive servers. In an array of small servers, the load balancing system is able to keep a group of servers in a state of readiness—threads loaded and running, memory allocated, database connections prepared—so that when the next function request comes, the small server is ready immediately.

Software engineers find debugging easier because fewer threads are running at any time. Large-scale multiprocessor systems manage and coordinate threads among the installed processors. Small, inexpensive servers run only the threads needed for the local Web-enabled application.

Company financial managers like the flapjacks architecture because they can buy lots of small, inexpensive servers and avoid the giant system purchases. The small server category has rapidly reached commodity status in the computer industry. The low-end server pricing gives financial managers power with server manufacturers to build and equip multiprocessor servers with large memory and hard-drive capacity at bargain prices.

Finally, network managers like flapjacks for the flexibility provided by all those small servers. As one IT manager put it, "If a system fails on any given morning, I can always go to the neighborhood giant computer supermarket and buy replacement parts." Additionally, small servers are easy to swap in and out of a rack of other equipment should they fail. During the swap, the load balancer directs traffic to other working servers.

Adopting Flapjacks and Intelligent Test Agents

While the flapjacks architecture and intelligent test agents are ideal for building scalable Web-enabled applications, that idealness may not be apparent to the various constituents at your business. Some selling may be required to get a new company to agree to a flapjacks/intelligent test agent server infrastructure strategy.

Table 2–5 considers the audience and their goals. Understanding how the Flapjacks architecture impacts the major groups at your company is key to successful adoption.

Table 2–5 Flapjacks Architecture Audience and Goals

Audience	Needs	Issues
Developers	Unit test new source code modules	Does the module return the correct values with the right state and input?
Quality Assurance technicians	Simulate real-world production-loaded environments.	What is the performance and scalability profile of the application? What features have regressed so previous fixed bugs are now present again?
IT Managers	Profile data center needs of a new Web-enabled application or an update to a Web-enabled application.	Does the datacenter still meet the application's needs? Also, use as a monitor for when the Web-enabled application fails.

Combining the flapjacks architecture and intelligent test agents is a good solution to meet each audience's needs. For example, the developer uses an intelligent test agent to bring an application back to a known state. The Quality Assurance (QA) manager uses an intelligent test agent to find fixed bugs that have regressed. And production managers use intelligent test agents to secure enough servers in a data center to support user needs.

Developers

The Internet software development lifecycle introduces a new concept in software testing: *rapid application testing*. Desktop applications and pre-Web-enabled application software insulated the software development team from direct responses from users. With the disintermediation between developer and user removed by the Internet, Web-enabled application publishers hear directly from users as errors happen. The users sound off with their email messages or with the mouse clicks that take them away from your Web-enabled application. In the Internet software development lifecycle, catching errors before users find them has a much higher priority to a business's success than ever before.

Developers used to the pre-Internet software development lifecycle are in for a shock when the first wave of user error reports arrive. Your company will expect near real-time solutions to user error reports. The developer who uses intelligent test agents can look back at agent logs to determine the source, and often the location, of a problem in the code. The developer who avoids intelligent test agents has only the Web-enabled application log files to review for help.

Consider the difference between log files created by a Web-enabled application server and an intelligent test agent.

```
127.0.0.1 - - [22/Jan/2002:00:12:41 -0600] "GET /mailman/
listinfo/oukids HTTP/1.1" 200 5358
127.0.0.1 - - [22/Jan/2002:00:12:41 -0600] "GET /icons/mail-
man.jpg HTTP/1.1" 200 2022
127.0.0.1 - - [22/Jan/2002:00:12:41 -0600] "GET /icons/gnu-
head-tiny.jpg HTTP/1.1" 200 3049
127.0.0.1 - - [22/Jan/2002:00:12:41 -0600] "GET /icons/
PythonPowered.png HTTP/1.1" 200 945
127.0.0.1 - - [22/Jan/2002:00:14:27 -0600] "GET / HTTP/1.1"
200 1489
```

This is a standard Apache-style log file. The file includes an entry for each request handled by the server. Imagine having only a log file to help you deduce and fix a server problem!

Intelligent test agents are modeled after the needs of an archetypal user—like Madeline from Chapter 1 or Ann from earlier in this chapter. The test agent log provides much more information to find and fix a bug.

```
Agent name, Task, Results, Module, Date:Time

Madeline, Sign-in, Ok, com.p2t.signin, 10:21:22:10
Steps taken: chose random user, lookup user id/password,
sign-in, all modules ran within tolerances

Madeline, Find product 7, OK, com.p2t.findit, 10:21:22:31
Steps taken: chose product previously ordered, find product
module notes: At 80% of catalog capacity warning

Madeline, Choose product 7, OK, com.p2t.formchk, 10:21:22:45
Steps taken: Check credit module warns of credit overrun
```

```
Madeline, Add quantity, OK, com.p2t2.quanchk, 10:21:23:15
Steps taken: Add product to shopping basket module. All mod-
ules run successfully

Madeline, Sub quantity, Fail, com.p2t2.subchck, 10:21:23:25
Steps taken: Subtract product from shopping basket module.
Shopping basket module returned invalid integer value 871
```

The Apache Web Server log file is click-stream oriented and answers the question: What requests did the Web-enabled application process? The intelligent test agent log is user-goal oriented and answers the question: Did the user reach his or her goal?

Looking at log files is important to prioritize your efforts to maintain Web-enabled applications. With intelligent test agents at work, summarizing the log file data into an immediately visible meter becomes possible. Figure 2–5 shows how such a meter may appear.

While intelligent agents provide valuable and timely data for a developer who needs to perform unit testing of new source code modules, intelligent test agents enable QA managers to simulate real-world production load environments, as you will see next.

Quality Assurance Technicians

The best QA technicians are a different kind of software developer. They keep the user's goals in mind and validate a Web-enabled application's ability to achieve these goals. Much of a QA technician's day may seem like tedium, but many QA technicians enjoy finding a problem and working with a developer to arrive at a solution. Many of the best QA technicians will show the developer the code causing the problem, so QA technicians must be expert at code, errors, and usability at the same time.

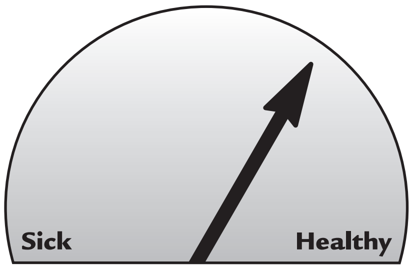


Figure 2–5 A live meter summarizes the results of running intelligent test agents to show at a glance the relative health of a Web-enabled application.

While the Internet software development lifecycle features a much stronger emphasis on code documentation than other non-Internet development lifecycles, the function of a software module is frequently a place where bugs and problems get past the QA technician. The developer makes changes to a module, writes change notes, and then releases the code to the QA technician. The notes are often open to various interpretations, which can lead to errors.

Imagine that a new module's change notes come with an intelligent test agent written by the developer. The test agent is much more powerful and accurate at describing the module than mere change notes. Consequently, the test agent becomes a more effective communication medium between developer and QA manager.

Information Technology Managers

IT managers maintain the system after the initial development is completed. IT is concerned with maintenance and monitoring.

Even if a Web-enabled application consisted of only a single line of code, the service will need maintenance over time. The job climate for Web-enabled application developers is driving many turnovers within businesses. Many times it's up to an IT manager to try to debug a problem in a Web-enabled application, even though the original developer has moved on. Imagine which scenario seems better to the IT manager: looking through source code comments to find a malfunctioning module or reading an intelligent test agent to see how the module should be performing? Intelligent test agents are a powerful means for a developer to show an IT manager the correct behavior of a Web-enabled application and software module.

Monitoring Web-enabled applications requires intelligent test agents. In the early days of the Internet, if a Web-enabled application returned a page, it could be assumed to be working. What happens when the returned page contains an error message or invalid data? An intelligent test agent understands the context of a Web-enabled application's functions and can sound an alert—through email, pagers, or in log files—to catch the IT manager's attention.

Intelligent test agents act as a communication medium between the developer, QA manager, and IT manager and are a hugely efficient way to develop and roll out Web-enabled applications. By focusing everyone's attention on intelligent test agents, all the team members know to expect intelligent test agent delivery with new code—this supports agile (Extreme/XP) software development's practice of testing first. Once an organization adopts test agents, all development projects will include test agents.

An organization's choice of test tools determines the likelihood that the team will consistently apply test agents. When a test tool is appreciated by QA technicians but not appreciated by developers, the consistent application of intelligent test agents will suffer.

Building Intelligent Test Agents in a Flapjacks Environment

Moving a Web-enabled application into a production environment requires assurances of high availability and consistently good performance. Test agents build these assurances from the data that they create. The choice of test tool, therefore, is important. Following is a recommended checklist that developers, QA managers, and IT managers should keep in mind when choosing tools with which to test a Web-enabled application:

- **Stateful testing**—When you use a Web-enabled application to set a value, does the server respond correctly later on?
- **Privilege testing**—What happens when the everyday user tries to access a control that is authorized only for administrators?
- **Speed testing**—Is the Web-enabled application taking too long to respond?
- **Boundary timing testing**—What happens when your Web-enabled application request times out or takes a really long time to respond?
- **Regression testing**—Did a new build break an existing function?

These are fairly common tests for any software application. A template for a test matrix to understand the performance characteristics of a Web-enabled application at various levels of load is shown in Table 2–6.

Many developers are skeptical when evaluating commercial test tools. The thinking goes: *Why should the company pay all this money for a tool with a new set of instructions or language to learn, when I could just write the test agent myself?*

Table 2–6 Test Matrix Template

Type of test	1 user	50 users	500 users
Stateful testing	❑	❑	❑
Privilege testing	❑	❑	❑
Speed testing	❑	❑	❑
Boundary timing testing	❑	❑	❑
Regression testing	❑	❑	❑

The problem with writing your own test tool comes down to one word: *maintenance*. Just like every other software application, the test tool will need to be maintained. Who will maintain the code when the test tool developer is no longer at the company?

A brief, unscientific survey of Silicon Valley developers who wrote their own test tools found that maintaining the tool grew from a minor irritation to a huge problem. The typical developer's first attempt at writing tools resulted in a fairly robust set of Java classes or Visual Basic methods that issue HTTP requests to a Web server and perform some simple analysis of the results. Writing a test agent consisted of writing Java code that sequentially called the correct test object.

For example, an agent that reads through an online discussion forum message base looks like this:

1. Read the first page of the site (which included URLs to the discussion messages).
2. Randomly read a discussion message.
3. If the message has a reply, read the reply message.
4. Repeat step 3 until there are no more reply messages.

The resulting test agent could be written as a small Java application. At some point, the format for the URL to read the first page of the site will change. This requires maintenance on the Java test code.

As a matter of fact, every new change to the Web-enabled application required some subtle change in the test agent. Each and every change brought the developer back to the test agent code. While the test objects—getting Web pages, signing in, testing for the correct return values—stayed the same from test agent to test agent, the calling sequence would be different.

Unfortunately, neither Java nor Visual Basic defines a scripting model as part of its design. In a scripting model, groups of functions are called in a particular sequence with branches occurring based on parameter evaluation. Using Java and Visual Basic leaves the programmer with the responsibility to invent a way to call objects in a particular sequence. As a result, many programmers invent a simple scripting language inside their code to implement the business rules or logic of the agent. Some more advanced applications even expose a scripting language to users. In the whole of computing, literally thousands of applications use their own scripting languages.

Looking at the test agent code, does it make sense to add a scripting language to assemble test agents from a library of test objects? The test objects perform the individual test routines that rarely need to change. And a scripting language would be used to assemble and control the parameters of the objects—a scripting language that would be easier to alter for different levels and types of Web-enabled application testing.

The benefit to all developers, QA technicians, and IT managers is that a programmer writes the more labor-intensive test objects only once and adapts the scripting language as needed. The scripting language enables the developers, QA technicians, and IT managers that are more comfortable with scripts than hard-core object-oriented code to write their own test agents. The scripts are readable by mere human beings and can easily be shared and swapped with others.

Modern scripting languages, such as Jython (details at www.jython.org) enable Web-enabled application test agents of the form described in this book. Scripting provides a common reference for the way test agent scripts, variables, and program flow are noted.

Script Languages and Test Agents

Most software tools mature to a point at which a scripting language makes sense. (Many times, a scripting language's appearance is a sign of a mature tool.) However, the downside to using scripting languages is that each one exists in its own little world. Consequently you may find yourself in a company where each group uses a different scripting language and tool. Table 2–7 illustrates just a very few of the popular choices.

Table 2–7 is just a tiny sample of the many scripting languages and tools in use today at companies that use Web-enabled applications. So your odds of

Table 2–7 Examples of Scripting Languages and Tools for Testing and Monitoring Web-Enabled Applications

Software developers	QA technicians	IT managers
Java objects	Python	Perl
jUnit—unit testing framework	OpenSTA	PHP
C#	TestMaker	HP OpenView management console
VB.NET	Ruby	Intermapper

finding a single scripting language and test tool that everyone will agree to use are small. However, there is a growing trend toward delivering scripting language functionality that runs on a common run-time environment. Jython is a good example of this trend.

Jython is an open source project that implements the popular Python programming language in Java. The Python script commands are compiled on-the-fly into Java byte-code. Java byte-codes are the intermediary language that Java uses internally to process commands on the local machine. The Java virtual machine executes the byte-codes on the local machine's processor. Byte-codes make it possible for Java applications to run on any machine where Java runs. As Figure 2–6 shows, this approach to using a common run-time environment is available to other scripting languages too.

A powerful advantage to using a common run-time environment is that the script language commands have access to the native script language objects

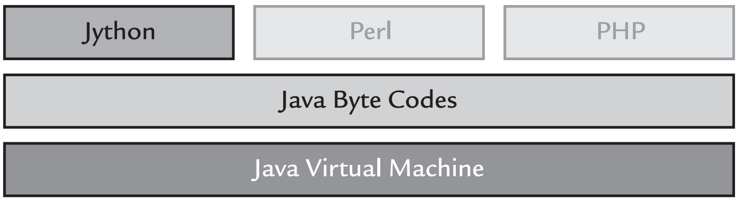


Figure 2–6 A growing trend is to have scripting languages use a common runtime environment. Jython is an example of what is possible. Jython scripts are compiled on-the-fly to Java byte-codes to be run on a Java virtual machine anywhere Java runs. With some work, Perl, PHP, and any other scripting language can use the same common run-time environment.

as well as to any public objects the run-time environment has available. In Jython's case, any Jython script can use all of Python's objects plus the script can access any Java object. This is a natural bridge between Jython scripters and Java object developers.

To see the power of Jython and Java objects at work, the following script is an example of a test agent script that uses both.

While you read the following script, keep in mind that if scripting or software code is not your interest, then test utilities, such as TestMaker, provide graphical interfaces and test agent recorder utilities to keep you from having to write script code. However, an understanding of the functions of a typical test agent script will provide you with a foundation to know what to expect from the test utility graphical interface.

```
# Agent name: Example Form Test
#
# Purpose:
# Calculates a Transaction-Per-Second index for a
# Web-enabled HTTP application

from com.pushtotest.tool.protocolhandler import \
  ProtocolHandler, Header, Body, HTTPProtocol, \
  HTTPBody, HTTPHeader
from com.pushtotest.tool.response import Response, \
  ResponseLinkConfig, SimpleSearchLink
from java.util import Date
from java.lang import Exception

# The test class implements the steps recorded while
# watching a user operate the application using a browser

class test:

    # Set-up
    def __init__(self):
        self.steps = 0          # Steps in transaction
        self.errors = 0         # Counter of errors
        self.passed = 0         # Value is 1 if no exceptions

    # Test
    def run(self):
        self.steps = 0;
        start_time = Date().time
        http = ProtocolHandler.getProtocol("http")
        # Request the welcome page with the html form

        http.setUrl( \
          '''http://examples.pushtotest.com/responder \
          /htmlresponder?file=file2.html''' )
        http.setType( HTTPProtocol.GET )
        response = http.connect( 0 )
        self.steps += 1
        code = response.getResponseCode()
        if code <> 200:
            print '''Error response code:''',self.code
            self.passed = 0
            self.errors += 1
        else:
            self.passed = 1

        # Submit the form values to the host

        http.setUrl( \
          '''http://examples.pushtotest.com/responder \
          /htmlresponder''')
        body = ProtocolHandler.getBody( 'http' )
        http.setBody( body )
        http.setType( HTTPProtocol.POST )
        body.addParameter('''firstname''', '''Frank''')
        body.addParameter('''lastname''', '''Cohen''')
        body.addParameter('''phone''', '''408 374 7426''')
        body.addParameter('''account''', '''10381838''')
        body.addParameter('''amount''', '''1838.12''')
        body.addParameter('''Transfer''', \
          '''Transfer Funds''')
        response = http.connect( 0 )
        self.steps += 1
        code = response.getResponseCode()
        if code <> 200:
            print '''Error response code:''',self.code
            self.passed = 0
            self.errors += 1
        else:
            self.passed = 1

    # Getter methods for various timing values
    def getTotaltime(self):
        return self.totaltime

    def getSteps(self):
        return self.steps
    def getTransactions(self):
        return self.transactions

    def getErrors(self):
        return self.errors

    def getPass(self):
        return self.passed

    # Clean-up
    def cleanup(self):
        print "Done."

# Transactions-per-second Index Test
# Run the test 100 times and then print the results

print "----------------------------------------------------"
print "Transactions-per-second Index Test"
print "brought to you by TestMaker from PushToTest"
print
print "----------------------------------------------------"
print "Starting test"

t = test()                # Create a test object
transactions = 0          # How many transactions completed
exCount = 0               # Count of exceptions thrown
count = 50                # Number of times to run the test

starttime = Date().time   # record the current time

print "Test running"
print

for i in range( count ):
    print "-",
print

for i in range( count ):
    try:
        t.run()
        transactions+=1
        print "-",
    except Exception, ex:
        print "Encountered an error:",ex
        exCount += 1

totaltime = Date().time - starttime
```

```
print
print
print "--------------------------------------------------------"
print "Results"

tps = float( transactions ) / float( totaltime ) * 100
print "Transactions Per Second (TPS): %9.2f" % tps
print "Total transactions processed: ", transactions
print "Total time to process all transactions: ", totaltime
print "Each transaction had", t.getSteps(),"steps"
print "Exceptions thrown during the test: ", exCount
print "Errors encountered during the test: ", t.getErrors()
print

t.cleanup()
```

Scripting languages like Jython make test agent logic much more apparent because of the straightforward style of the script's formatting. The underlying test objects handle the actual communication with the host service, while the scripting language worries about the logic of the test agent.

The example just given makes use of the scripting language and library of test objects found in TestMaker, an open source utility for testing Web-enabled applications for performance and scalability. TestMaker and all the examples in this book are available for free download at http://www.push-totest.com and licensed under an Apache-style free open source license.

To better understand the power of a scripting language when used to build test agents, look at this test agent script in more detail:

```
from com.pushtotest.tool.protocolhandler import \
  ProtocolHandler, Header, Body, HTTPProtocol, \
  HTTPBody, HTTPHeader
from com.pushtotest.tool.response import Response, \
  ResponseLinkConfig, SimpleSearchLink
from java.util import Date
from java.lang import Exception
```

The import statement identifies the library of test objects to the script language compiler. The test object library will do the heavy lifting of communicating with the Web-enabled application host. For example, `HTTPProtocol` is an object that handles communication to a Web host through the HTTP protocol. Additionally, this script will use Java's Date object and report errors using Java's Exception object.

```
class test:

    # Set-up
    def __init__(self):
        self.steps = 0          # Steps in transaction
        self.errors = 0         # Counter of errors
        self.passed = 0         # Value is 1 if no exceptions
```

The script uses Jython's object-oriented methods to create a new class named test. The test class initializes three new variables: steps, errors, and passed. The steps variable keeps track of how many HTTP requests are used in each transaction. In this example, a transaction is made up of two steps: get a Web page containing a form, and then submit the form values to the host.

```
def run(self):
        self.steps = 0;
        start_time = Date().time
        http = ProtocolHandler.getProtocol("http")
```

The test class has a single method named run. This method tests the host by completing a single transaction. We will see further on how the test agent script calls the run method multiple times to learn the system's average throughput.

The run method begins by using the `ProtocolHandler` object provided in TestMaker to get a new HTTP protocol handler. The script establishes a new variable named http that points to a new HTTP protocol handler object. The protocol handler will communicate with a Web server using HTTP to get a page with an HTML form. Use your browser to look at the following URL to understand this Web-enabled application better: http://examples.pushtotest.com/responder/htmlresponder?file=file2.html

```
        http.setUrl( \
          '''http://examples.pushtotest.com/responder \
          /htmlresponder?file=file2.html''' )
        http.setType( HTTPProtocol.GET )
        response = http.connect( 0 )
        self.steps += 1
        code = response.getResponseCode()
        if code <> 200:
            print '''Error response code:''',self.code
            self.passed = 0
            self.errors += 1
        else:
            self.passed = 1
```

The script tells the http object to use the provided URL and to use the HTTP Get protocol. The http object provides a `connect` method that returns a `response` object. The `response` object contains the HTML codes received from the host, the HTTP header values, and any cookie values. The `response` object provides many methods to validate the response data, including a `getResponseCode` method to determine the results result code received from the host. The HTTP protocol defines a response code of 200 as a normal response. In the event of a response code over 200, the script prints the response code to the output panel.

```
        http.setUrl( \
          '''http://examples.pushtotest.com/responder \
          /htmlresponder''')
        body = ProtocolHandler.getBody( 'http' )
        http.setBody( body )
        http.setType( HTTPProtocol.POST )
        body.addParameter('''firstname''', '''Frank''')
        body.addParameter('''lastname''', '''Cohen''')
        body.addParameter('''phone''', '''408 374 7426''')
        body.addParameter('''account''', '''10381838''')
        body.addParameter('''amount''', '''1838.12''')
        body.addParameter('''Transfer''', \
          '''Transfer Funds''')
        response = http.connect( 0 )
```

The next group of script commands perform a `POST` command to the Web-enabled application host. This is equivalent to a user filling in the form elements on a Web page and clicking Submit.

```
# Getter methods for various timing values
    def getTotaltime(self):
        return self.totaltime

    def getSteps(self):
        return self.steps

    def getTransactions(self):
        return self.transactions

    def getErrors(self):
        return self.errors

    def getPass(self):
        return self.passed
    # Clean-up
    def cleanup(self):
        print "Done."
```

These commands add several helper methods to the test class. Specifically, they provide methods for the test class to return the test results data to another class. We will see these used in the next section of code. That concludes the script codes needed to create the test class. Now we are ready to use the test class.

```
t = test()                # Create a test object
```

The test script instantiates a test object that will be referenced using the t variable. That's how easy Jython is to create objects!

```
transactions = 0          # How many transactions completed
exCount = 0               # Count of exceptions thrown
count = 50                # Number of times to run the test
starttime = Date().time   # record the current time
```

The test script establishes a few variables that will be used later to report the transactions-per-second throughput of the host.

```
for i in range( count ):
    try:
        t.run()
        transactions+=1
        print "-",
    except Exception, ex:
        print "Encountered an error:",ex
        exCount += 1
```

The test script uses the `for` command to repeat the block of code that calls the run method in the test object. Recall that the count variable was set to 50, so the test agent will run the test class 50 times.

If you are wondering about the formatting of the test script, Jython uses spacing to delimit blocks of code. Spacing is important. If you want to jump into using this script now, I recommend you download the script from the PushToTest Web site at http://www.pushtotest.com/ptt to save yourself the time of having to type in the agent script code.

```
tps = float( transactions ) / float( totaltime ) * 100
print "Transactions Per Second (TPS): %9.2f" % tps
print "Total transactions processed: ", transactions
print "Total time to process all transactions: ", totaltime
print "Each transaction had", t.getSteps(),"steps"
print "Exceptions thrown during the test: ", exCount
print "Errors encountered during the test: ", t.getErrors()
```

Finally, the agent script calculates and prints the throughput of the test in a transactions-per-second report.

Generating Meaningful Data

Intelligent test agents test a Web-enabled application by operating the software just as a user would. To the Web-enabled application, the requests from a test agent appear to be legitimate users. The Web-enabled application works on the request and returns data. Multiple concurrently running test agents simulate real-world production environments where many users are accessing a system at the same time. All this activity creates raw data in these forms:

- The test agent creates a log to store details of the requests, including how long it took to set up and process a request and how many errors were returned.
- The Web-enabled application host creates a log showing how many requests were processed and details on the results, including timeouts and dropped connections.
- The Web-enabled application server hardware includes an operating system agent (usually using the Simple Node Monitor Protocol, or SNMP) to store usage information, including network bandwidth, hard-drive space, CPU usage, and memory usage.
- The network routers and other infrastructure create logs to track resource allocation and services used.

Each of these sources of raw data may be turned into meaningful data using intelligent test agent technology. The key to making a lot of raw data meaningful is to define the parameters of a well-performing system. The Web-enabled Applications Points System is a good methodology. Finally, meaningful data is generated when automation technology completes the tasks necessary to monitor and observe good or poor performance in a Web-enabled application.

Summary

Improvements in Internet technology, bandwidth, and service levels have powered the phenomenal growth in Web-enabled application development projects. The modern Web-enabled application development team includes developers who have reached entirely new levels of productivity by using new software development tools, new software development lifecycles, and new methodologies to write software. Software testing tools, techniques, and methods have evolved, too. Developers, QA technicians, and IT managers find software testing must be intelligent or the testing results will be meaningless, or worse, misleading. The new software test techniques help developers prioritize their efforts to maximize the user's ability to achieve important goals.

New tools introduced in this chapter, for example, the Web rubric and Web-enabled Application Points System, provide easy-to-follow criteria to determine a Web-enabled application's health. New data center frameworks and an open source test automation tool, TestMaker, provide easy and efficient ways to build Web-enabled applications that scale and perform in real-world production environments today.

The choice of tools, infrastructure, and methodologies makes a significant difference on a test agent's ability to locate problems and monitor Web-enabled applications for high service levels. Experience shows us that when a modern Web-enabled application development team has reached these goals for its test tools, the Web-enabled application will also meet user goals.

- Software developers, QA technicians, and IT managers need to have access to the appropriate tools regardless of cost or corporate standards.
- Test agents need to be intelligent.
- Testing Web-enabled applications and SOAP-based Web services requires a test to drive the application intelligently, as a normal everyday user would.
- Test agents need to test systems for scalability and performance under near-production loads.
- Test tools need to be used.

In this chapter, you learned how to quantify good system performance in systems that integrate Web-enabled applications. In Chapter 3, you will see how tests for functionality, scalability, and performance are linked to user success.

3
Modeling Tests

"I don't test," a friend told me. It is not because she was against testing. She would say the same thing about software design and architecture. From her perspective, testing is an obscure and difficult-to-learn process only practiced by test professionals. If you could not tell already, this book takes an *everyone tests* philosophy. I believe that everyone—users, administrators, software developers, architects, quality assurance technicians, network managers, and management—tests each time they use a Web-enabled application.

The trick is to find a way to turn these ad-hoc tests into useful and meaningful test data in a distributed and interoperating Web-enabled application environment. Useful and meaningful test data greatly improve how we prioritize which bugs and problems get solved first, identifies where to spend our budgets on maintenance and improvements, and draws us closer to delivering user success.

To produce meaningful test data, your tests must determine how close the user gets to meeting their needs while using the software. This chapter shows how to model user behavior and implement that behavior in an intelligent test agent. Intelligent test agents are the most powerful way to automate test suites. Understanding the test environment and test tools for building intelligent test agents is necessary to understand test suite results.

The previous chapters set the stage by describing the ingredients needed to perform meaningful tests of Web-enabled applications; this chapter raises the curtain to reveal how the players—servers, routers, data centers, Internet service providers (ISPs), and software—fill their roles in delivering highly available and well-performing Web-enabled applications. Rather than describe a single method, this chapter describes methodologies that can be applied to many testing scenarios and shows how human nature plays a significant role in deciding infrastructure requirements and test methodology.

Modeling User Behavior for Meaningful Test Results

Businesses need answers to questions about building, deploying, and maintaining a Web-enabled application and test methodology and strategy provides answers. For example, a business may ask, "How many more servers should I buy to host the forecast of new users and customers?" Spreadsheets and models can offer a head start on forecasting datacenter needs; however, eventually models will need hard data to back up any assumptions. The answers to the questions raised by a model usually depend on data that describes the operating environment—the data center, user interface, and technology deployed. Testing Web-enabled applications provides meaningful answers to sizing, bandwidth, and functionality questions.

Businesses developing and delivering Web-enabled applications run into the same set of questions. Table 3–1 lists the questions I most often need to answer.

Table 3–1 Questions to Ask When Developing Web-Enabled Applications

Question asked	Data needed	Test solution to apply
While the data center is built and the usage forecast is set, is your data center really ready?	Testing to determine data center readiness	Use intelligent test agents to determine the Scalability and Performance Index (SPI) for the Web-enabled application. The SPI (described later in this chapter) measures data center readiness.
What components and bandwidth will the data center need?	Testing to estimate data center needs	Web-enabled application simulation using intelligent test agents will give early answers to aim for. (Test agents are described later in this chapter.)

Table 3–1 Questions to Ask When Developing Web-Enabled Applications

When users put an item in a shopping basket, is it still there an hour later? Did the item not appear in your shopping basket, but instead appear in another user's shopping basket? Did the system allow this privilege error?	Testing to find state and boundary problems	Unit testing with intelligent agents is good at probing a Web-enabled application function with both valid and invalid data. The results show that parts of a Web-enabled application are not functioning correctly. Intelligent agents automate unit tests to streamline testing and reduce testing costs.
How will the Web-enabled application operate when higher-than-expected use is encountered?	Testing to be prepared for higher-than-expected volumes	A network of intelligent test agents running concurrently will show how the Web-enabled application operates during periods of intense overuse.
As software is maintained, old bugs may find new life. What was once fixed and is now broken again?	Testing to find software regression	Intelligent test agents monitor by stepping a Web-enabled application through its functions. When new software is available the monitor tests that previously available functions are still working.

Testing Web-enabled applications plays an important role in solving business issues for a company. By recognizing how tests can solve business issues, the test professional learns valuable answers to important questions.

Over the years I learned that the highest quality Web-enabled application systems were designed to be tested and maintained. Good system design prepares for issues such as these:

- How frequently will components fail? Are replacement parts on hand? What steps are needed to replace a component?
- What are the expected steps needed to maintain the system? For example, a data-intensive Web-enabled application will need index and tables data to be re-created to capture unused disk space and memory. Will the system be available to users while an index is rebuilt?

- Where will new components be added to the system? Will more physical space be needed to accommodate new computer hardware? Where will new software be installed?
- What areas are expected to get better if occasionally reviewed for efficiency and performance? We can expect improvements in memory, CPU, and storage technology. Should the system be planned to incorporate these improvements?

Understanding the lifecycle for developing Web-enabled applications is integral to answering business questions and preparing for maintenance.

Lifecycles, Projects, and Human Nature

Human nature plays a significant role in deciding infrastructure requirements and test methodology. As humans we base our decisions on past experience, credibility, an understanding of the facts, the style with which the data is presented, and many other factors. We need to keep human nature in mind when designing a product lifecycle, new architecture, and a test. For example, consider being a network manager at a transportation company. The company decides to use a Web-enabled application to publish fare and schedule information currently hosted on an established database-driven system and accessed by users through a call center. The company needs to estimate the number of servers to buy and Internet bandwidth for its data center. As the network manager, imagine presenting test result data that was collected in a loose and ad-hoc way to a senior manager that has a rigid and hierarchical style.

By understanding business management style, we can shape a test to be most effective with management. Later in this section we define four types of management styles and their impact on design and testing.

In my experience, the most meaningful test data comes from test teams that use a well-understood software development lifecycle. Web-enabled application software development is managed as a project and developed in a lifecycle. Project requirements define the people, tools, goals, and schedule. The lifecycle describes the milestones and checkpoints that are common to all Web-enabled application projects.

Web-enabled applications have borrowed from traditional software development methods to form an Internet software development lifecycle. The immediacy of the user—they're only an email message away—adds special twists to traditional development lifecycles. Here is a typical Internet software development lifecycle:

1. Specify the program from a mock-up of a Web site.
2. Write the software.
3. Unit test the application.
4. Fix the problems found in the unit test.
5. Internal employees test the application.
6. Fix the problems found.
7. Publish the software to the Internet.
8. Rapidly add minor bug fixes to the live servers.

Little time elapses between publishing the software to the Internet in step 8 and receiving the first feedback from users. Usually the feedback compels the business to address the user feedback in rapid fashion. Each change to the software sparks the start of a new lifecycle.

The lifecycle incorporates tasks from everyone involved in developing a Web-enabled application. Another way to look at the lifecycle is to understand the stages of development shown here:

- Write the requirements.
- Validate the requirements.
- Implement the project.
- Unit test the application.
- System test the application.
- Pre-deploy the application.
- Begin the production phase.

Defining phases and a lifecycle for a Web-enabled application project may give the appearance that the project will run in logical, well conceived, and proper steps. If only the senior management, users, vendors, service providers, sales and marketing, and financial controllers would stay out of the way! Each of these pull and twist the project with their special interests until the project looks like the one described in Figure 3–1.

The best-laid plans usually assume that the development team members, both internal and external, are cooperative. In reality, however, all these constituents have needs and requirements for a Web-enabled application that must be addressed. Many software projects start with well-defined Web-enabled appli-

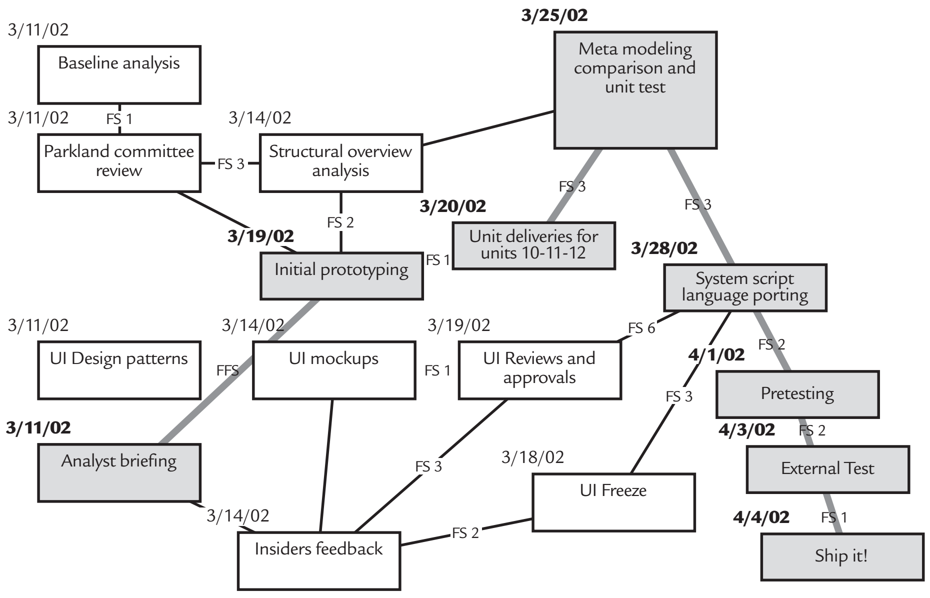


Figure 3–1 Managing the complex and interrelated milestones for development of a typical Web-enabled application has an impact on how software development teams approach projects.

cation project phases, but when all the project requirements are considered, the project can look like a tangled mess (Figure 3–1).

Confronted with this tangle of milestones and contingencies, software project managers typically separate into two camps concerning the best method to build, deploy, and maintain high-quality Web-enabled applications. One camp focuses the project team's resources on large-scale changes to a Web-enabled application. New software releases require a huge effort leading to a single launch date. The other camp focuses its resources to "divide and conquer" a long list of enhancements. Rather than making major changes, a series of successive minor changes are developed.

Software project managers in enterprises hosting Web-enabled applications that prefer to maintain their software by constantly adding many small improvements and bug fixes over managing toward a single, comprehensive new version put a lot of stress on the software development team. The Micromax Lifecycle may help.

The Micromax Lifecycle

Micromax is a method used to deploy many small improvements to an existing software project. Micromax is used at major companies, such as Symantec and Sun Microsystems, with good results. Micromax defines three techniques: a method to categorize and prioritize problems, a method to distribute assignments to a team of developers, and automation techniques to test and validate the changes. Project managers benefit from Micromax by having predictable schedules and good resource allocation. Developers benefit from Micromax because the projects are self-contained and give the developer a chance to buy-in to the project rather than being handed a huge multifaceted goal. QA technicians benefit by knowing the best order in which to test and solve problems.

Categorizing Problems

Micromax defines a method for categorizing and prioritizing problems. Users, developers, managers, and analysts may report the problems. The goal is to develop metrics by which problems can be understood and solved. The more input the better.

Problems may also be known as bugs, changes, enhancement requests, wishes, and even undocumented features. Choose the terminology that works best for your team, including people outside the engineering group. A problem in Micromax is a statement of a change that will benefit users or the company. However, a problem report is categorized according to the effect on users. Table 3–2 describes the problem categories defined by Micromax.

Table 3–2 Micromax Problem Categories

Category	Explanation
1	Data loss
2	Function loss
3	Intermittent function loss
4	Function loss with workaround
5	Speed loss
6	Usability friction
7	Cosmetic

Category 1 problem reports are usually the most serious. Everyone wants to make his mark on life and seldom does a person want his marks removed. When an online banking Web-enabled application loses your most recent deposits, when the remote file server erases a file containing an important report, or even when a Web-enabled application erases all email messages when it was only supposed to delete a single message, that is a Category 1 problem.

Categories 2, 3, and 4 apply to features or functions in a Web-enabled application that do not work. The Web-enabled application will not complete its task—Category 2—or the task does not complete every time—Category 3—or the function does not work but there is a defined set of other steps that may be taken to accomplish the same result—Category 4.

Category 5 identifies problems in which a Web-enabled application function completes its task, but the time it takes is unacceptable to the user. Experience shows that every Web-enabled application defines acceptable times differently. A Web-enabled application providing sign-in function for live users likely has a 1- to 2-second acceptable speed rating. The same sign-in that takes 12 to 15 seconds is likely unacceptable. However, a Web-enabled application providing a chemical manufacturer with daily reports would accept a response time measured in seconds or minutes, because report viewers don't need up-to-the-second updates. Category 5 earned its place in the category list mostly as a response to software developers' usual behavior of writing software functions first and then modifying the software to perform quickly later.

Categories 6, 7, and 8 problems are the most challenging to identify. They border on being subjective judgment calls. For every wrongly placed button, incomprehensible list of instructions on the screen, and function that should be there but is strangely missing is a developer who will explain, with all the reason in the world, why the software was built as it is. Keep in mind the user goals when categorizing problems.

Category 6 identifies problems in which the Web-enabled application adequately completes a task; however, the task requires multiple steps, requires too much user knowledge of the context, stops the user cold from accomplishing a larger task, or is just the biggest bonehead user-interface design ever. Software users run into usability friction all the time. Take, for example, the printer that runs out of paper and asks the user whether she wants to "continue" or "finish." The user goal is to finish, but she needs to continue after adding more paper to the printer. Category 6 problems slow or prevent the user from reaching her goals.

Category 7 identifies problems involving icons, color selections, and user interface elements that appear out of place. Category 8 problems are observed when users complain that they have not reached their goals or are uncertain how they would use the Web-enabled application.

The Micromax system puts software problems into eight levels of inoperability, misuse, difficult interfaces, and slow performance—none of these is much fun, nor productive, to the user.

Prioritizing Problems

While problem categories are a good way to help you understand the nature of the Web-enabled application, and to direct efforts on resolutions, such categories by themselves may be misleading. If a Web-enabled application loses data for a single user but all the users are encountering slow response time, something in addition to categorization is needed. Prioritizing problems is a solution. Table 3–3 describes the problem priority levels defined by Micromax.

Table 3–3 Micromax Problem Priority Ratings

Priority level	Description
1	Unacceptable business risk
2	Urgent action needed for the product's success
3	Problem needs solution
4	Problem needs solution as time permits
5	Low risk to business goals

A problem priority rating of 1 indicates that serious damage, business risk, and loss may happen—I've heard it described as "someone's hair is on fire right now." A solution needs to be forthcoming or the company risks a serious downturn. For example, a Web-enabled application company that spent 40 percent of its annual marketing budget on a one-time trade conference may encounter a cosmetic (category 7) problem but set its priority to level 1 to avoid ridicule when the company logo does not display correctly in front of hundreds of influential conference attendees.

The flip side is a problem with a priority level 5. These are the problems that usually sit in a little box somewhere called "inconsequential." As a result, they are held in place in the problem tracking system but rarely go away—which is not necessarily a bad thing, because by their nature they pose little risk to the company, product, or user.

Reporting Problems

In Micromax, both user and project manager categorize problems. Project managers often find themselves arguing with the internal teams over the priority assignments in a list of bugs. "Is that problem really that important to solve now?" is usually the question of the moment.

Micromax depends on the customer to understand the categories and to apply appropriate understanding of the problem. Depending on users to categorize the problems has a side benefit in that the users' effort reduces the time it takes for the team to internalize the problem. Of course, you must give serious consideration to the ranking levels a user may apply to make sure there is consistency across user rankings. The project manager sets the category for the problem and has the users' input as another data point for the internal team to understand.

Criteria for Evaluating Problems

With the Micromax system in hand, the project manager has a means to categorize and prioritize bugs. The criteria the manager uses are just as important. Successful criteria accounts for the user goals on the Web-enabled application. For example, a Web-enabled application providing a secure, private extranet for document sharing used the following criteria to determine when the system was ready for launch:

- No category 1 or 2 problems with a priority of 1, 2, 3, or 4
- No category 1, 2, 3, 4, or 5 problems with a priority of 1, 2, or 3
- No category 6, 7, or 8 problems with a priority of 1 or 2

In this case, usability and cosmetic problems were acceptable for release; however, data loss problems were not acceptable for release.

In another example, a TV talk show that began syndicating its content using Web-enabled applications has more business equity to build in its brand than in accurate delivery of content. The release criteria looked like this:

- No category 7 problems with a priority of 1, 2, 3, or 4
- No category 6 or 7 problems with a priority of 1 or 2
- No category 1, 2, or 3 problems with a priority of 1 or 2
- No category 5 problems with a priority of 1, 2, or 3

In this example, the TV talk show wanted the focus to be on solving the cosmetic and speed problems. While it also wanted features to work, the focus was on making the Web-enabled application appear beautiful and well designed.

The Micromax system is useful to project managers, QA technicians, and development managers alike. The development manager determines criteria for assigning problems to developers. For example, assigning low priority problems to a developer new to the team reduces the risk of the developer making a less-than-adequate contribution to the Web-enabled application project. The criteria also define an agreement the developer makes to deliver a function against a specification. As we will see later in this chapter, this agreement plays an important role in unit testing and agile (also known as Extreme Programming, or XP) development processes.

Micromax is a good tool to have when a business chooses to improve and maintain a Web-enabled application in small increments and with many developers. Using Micromax, schedules become more predictable, the development team works closer, and users will applaud the improvements in the Web-enabled applications they use.

Considerations for Web-Enabled Application Tests

As I pointed out in Chapter 2, often the things impacting the functionality, performance, and scalability of your Web-enabled application has little to do with the actual code you write. The following sections of this chapter show what to look for, how to quantify performance, and a method for designing and testing Web-enabled applications.

Functionality and Scalability Testing

Businesses invest in Web-enabled applications to deliver functions to users, customers, distributors, employees, and partners. In this section, I present an example of a company that offers its employees an online bookstore to distribute company publications and a discussion of the goals of functionality

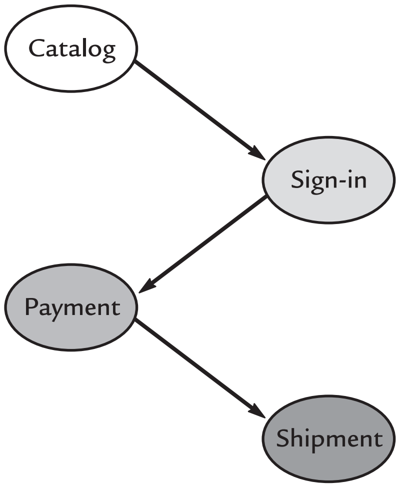


Figure 3–2 Individual services combined to make a system.

and scalability test methods. I then show how the system may be tested for functionality and then scalability.

Figure 3–2 shows the system design. The company provides a single integrated and branded experience for users while the back-end system is composed of four Web-enabled applications.

The online bookstore example uses a catalog service to look up a book by its title. Once the user chooses a book, a sign-in service identifies and authorizes the user. Next, a payment service posts a charge to the user's departmental budget in payment for a book. Finally, a shipment service takes the user's delivery information and fulfills the order.

To the user, the system appears to be a single application. On the backend, illustrated in Figure 3–3, the user is actually moving through four completely independent systems that have been federated to appear as a single Web-enabled application. The federation happens at the branding level (colors, logos, page layout), at the security level to provide a single sign-on across the whole application, and at the data level where the system shares data related to the order. The result is a consistent user experience through the flow of this Web-enabled application.

Users begin at the catalog service. Using a Web browser, the user accesses the search and selection capabilities built into the catalog service. The user selects a book and then clicks the "Order" button. The browser issues an HTTP `Post` command to the sign-in service. The `Post` command includes

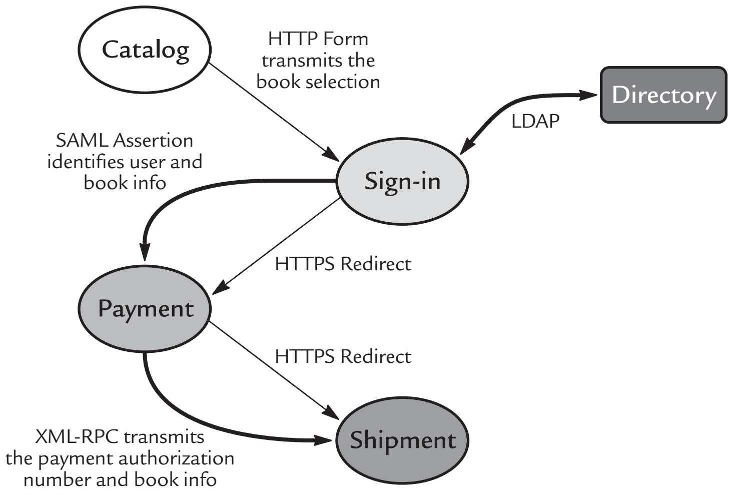

Catalog
HTTP Form transmits the book selection
Directory
LDAP
SAML Assertion identifies user and book info
Sign-in
HTTPS Redirect
Payment
HTTPS Redirect
XML-RPC transmits the payment authorization number and book info
Shipment

Figure 3–3 The bookstore example uses the SAML. the XML Remote Procedure language (XML-RPC), LDAP, and other protocols to federate the independent systems into one consistent user experience.

form data containing the chosen book selection. The sign-in service presents a Web page asking the user to type in their identity number and a password. The sign-in service makes a request to a directory service using the LDAP. LDAP is a popular authentication protocol based on the powerful X509 security standard. The directory service responds with an employee identification number. With a valid employee identification number, the sign-in service redirects the user's browser to the payment service and concurrently makes a SAML assertion call to the payment server.

Up until now the user's browser has been getting redirect commands from the catalog and sign-in service. The redirect commands put the transaction data (namely the book selected) into the URL. Unfortunately, this technique is limited by the maximum size of a URL the browser will handle. An alternative approach uses HTTP redirect commands and asynchronous requests between the services. The book identity, user identity, accounting information, and user shipping information move from service to service with these asynchronous calls and the user's browser redirects from service to service using a session identifier (like a browser cookie).

Often services support only a limited number of protocols, so this example has the sign-up and payment service using SAML and the shipping service using XML-RPC. Once the user identifies their payment information, the payment service redirects the user's browser to the shipment service and makes an XML-RPC call to the shipment service to identify the books ordered.

Looking at this system makes me wonder: *How do you test this system?* An interoperating system, such as the bookstore example, needs to work seamlessly every time. Testing for functionality will provide us with meaningful test data on the ability of the system to provide a seamless experience. Testing for scalability will show us that the system can handle groups of users of varying sizes every time.

Functional Testing

Functional tests are different than scalability and performance tests. Scalability tests answer questions about how functionality is affected when increasing numbers of users are on the system concurrently. Performance tests answer questions about how often the system fails to meet user goals. Functional tests answer the question: "Is the entire system working to deliver the user goals?"

Functional testing guarantees that the features of a Web-enabled application are operating as designed. The content the Web-enabled application returns is valid, and changes to the Web-enabled application are in place. For example, consider a business that uses resellers to sell its products. When a new reseller signs up, a company administrator uses a Web-enabled application to enable the reseller account. This action initiates several processes, including establishing an email address for the reseller, setting up wholesale pricing plans for the reseller, and establishing a sales quota/forecast for the reseller. Wouldn't it be great if there were one button the administrator could click to check that the reseller email, pricing, and quota are actually in place? Figure 3–4 shows just such a functional test.

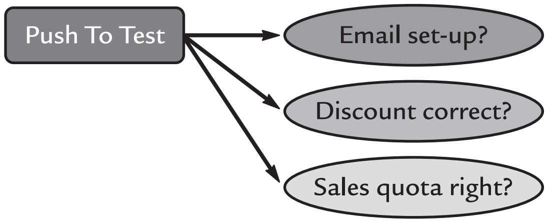


Figure 3–4 Click one button to test the system set-up.

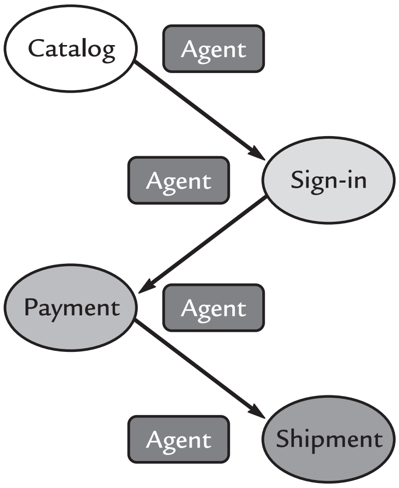


Figure 3–5 Intelligent test agents provide functional tests of each service to show an IT manager where problems exist.

In the bookstore example, a different type of functional testing is needed. Imagine that four independent-outsourcing companies provided the bookstore backend services. The goal of a functional test in that environment is to identify the source of a system problem as the problem happens. Imagine what any IT manager must go through when a deployed system uses services provided from multiple vendors. The test agent technology shown in Figure 3–5 is an answer.

Figure 3–5 shows how intelligent test agents may be deployed to conduct functional tests of each service. Test agents monitor each of the services of the overall bookstore system. A single console coordinates the activities of the test agents and provides a common location to hold and analyze the test agent data.

These test agents simulate the real use of each Web-enabled application in a system. The agents log actions and results back to a common log server. They meter the operation of the system at a component level. When a component fails, the system managers have test agent monitored data to uncover the failing Web-enabled application. Test agent data works double-duty because the data is also proof of meeting acceptable service levels.

Scalability Testing

Until now the bookstore test examples have checked the system for functionality by emulating the steps of a single user walking through the functions

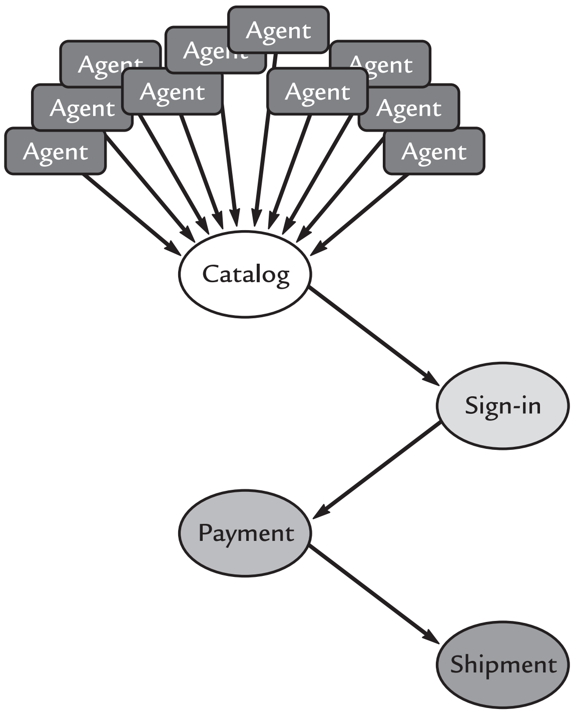


Figure 3–6 Using test agents to conduct scalability tests.

provided by the underlying services. Scalability testing tells us how the system will perform when many users walk through the system at the same time. Intelligent test agent technology is ideal for testing a system for scalability, as shown in Figure 3–6.

In this example, the test agents created to perform functionality tests are reused for a scalability test. The test agents implement the behavior of a user by driving the functions of the system. By running multiple copies of the test agents concurrently, we can observe how the system handles the load by assigning resources and bandwidth.

Testing Modules for Functionality and Scalability

Another way to understand the system's ability to serve users is to conduct functionality and scalability tests on the modules that provide services to a Web-enabled application. Computers serving Web-enabled applications become nodes in a grid of interconnected systems. These systems are efficiently designed around many small components. I am not saying that the

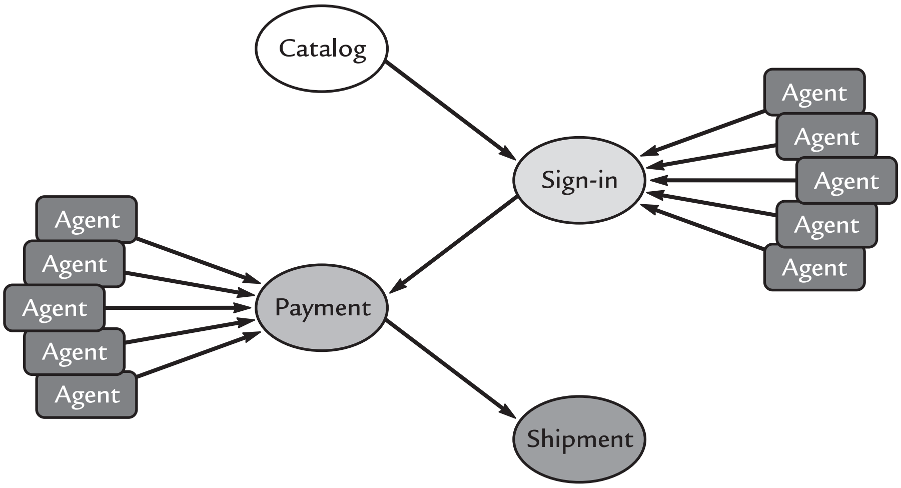


Figure 3–7 Functionality and scalability testing in a flapjacks environment enables us to test the modules that make up a system. The test agents use the native protocols of each module to make requests, and validate and measure the response to learn where bottlenecks and broken functions exist.

old-style large-scale mainframes are history, rather they just become one more node in the grid. That leaves us with the need to determine the reliability of each part of the overall system.

The flapjacks architecture, introduced in Chapter 2, is a Web-enabled application hosting model wherein a load balancer dispatches Web-enabled application requests to an application server. There, the flapjacks architecture provides us with some interesting opportunities to test modules for functionality and scalability. In particular, rather than testing the system by driving the test from the client side, we can drive tests at each of the modules in the overall system, as illustrated in Figure 3–7.

The flapjacks architecture uses standard modules to provide high quality of service and low cost. The modules usually include an application server that sits in front of a database server. The load balancer uses cookies to manage sessions and performs encryption/decryption of Secure Sockets Layer (SSL) secured communication. Testing Web-enabled application systems hosted in a flapjacks datacenter has these advantages:

- The load balancer enables the system to add more capacity dynamically, even during a test. This flexibility makes it much

easier to calculate the SPI, introduced in Chapter 3, for the system at various levels of load and available application servers. In addition, the application servers may offer varied features, including an assortment of processor configurations and speeds and various memory and storage configurations.
- Web-enabled applications deployed on intranets—as opposed to the public Internet—typically require authentication and encryption and usually use digital certificates and sometimes public key infrastructure (PKI). Testing intranet applications in a flapjacks environment allows us to learn the scalability index of the encryption system in isolation from the rest of the system.
- Using load balancers and IP layer routing—often using the Border Gateway Protocol (BGP)—enables the entire data center to become part of a network of data centers by using the load balancer to offload traffic during peak load times and to survive connectivity outages. Testing in this environment enables us to compare network segment performance.

Taking a different perspective on a Web-enabled application yields even more opportunities to test and optimize the system. The calling stack to handle a Web-enabled application request provides several natural locations to collect test data. The calling stack includes the levels described in Figure 3–8.

As a Web-enabled application request arrives, it passes through the firewall, load balancer, and Web server. If it is a SOAP-based Web Service request, then the request is additionally handled by a SOAP parser, XML parser, and various serializers that turn the request into objects in the native platform and language. Business rules instruct the application to build an appropriate response. The business objects connect to the database to find stored data needed to answer the request. From the database, the request returns all the way up the previous stack of systems to eventually send a response back to the requesting application. Each stage of the Web-enabled application request stack is a place to collect test data, including the following:

- **Web server**. Most Web servers keep several log files, including logs of page requests, error/exception messages, and servlet/COM (component object model) object messages. Log locations and contents are largely configurable to some extent.

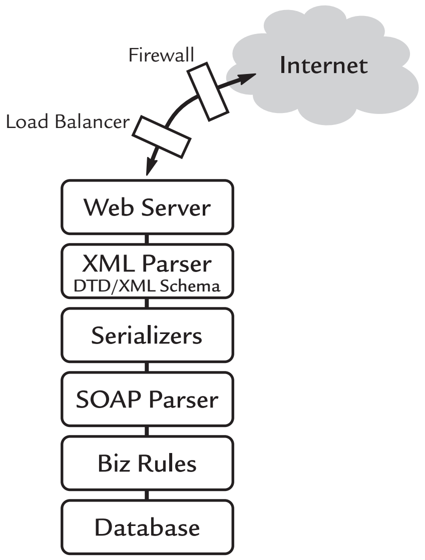


Figure 3–8 The call path for a typical Web-enabled application shows us many places where we may test and optimize for better scalability, functionality, and performance.

- **XML parser.** The SOAP parser handles communication to the Web-enabled application host, while the XML parser does the heavy lifting of reading and validating the XML document.
- **SOAP parser.** Application servers such as BEA WebLogic and IBM WebSphere include integrated SOAP parser libraries so the SOAP parser operating data is found in the application server logs. On the other hand, many Web-enabled applications run as their own application server. In this case, the SOAP parser they bundle—Apache Axis, for example—stores operating data in a package log.
- **Serializers.** Create objects native to the local operating environment from the XML codes in the request. Serializers log their operating data to a log file.
- **Business rules.** Normally implemented as a set of servlets or Distributed COM (DCOM) objects and are run in servlet or DCOM containers such as Apache Tomcat. Look into the application log of the application server.
- **Database.** Database servers maintain extensive logs on the local machine of their operation, optimizations, and other tools.

The downside to collecting all this test data is the resulting sea of data. All that data can make you feel like you are drowning! Systems that integrate several modules, such as the bookstore example above, generate huge amounts of result data by default. The subsystems used by Web-enabled applications include commercial and open source software packages that create log files describing actions that occurred. For example, an application server will log incoming requests, application-specific log data, and errors by default. Also, by default, the log data is stored on the local file system. This can be especially problematic in a Web-enabled application environment, where portions of the system are inaccessible from the local network.

Many commercial software packages include built-in data-collecting tools. Tools for collecting and analyzing simple Web applications (HTTP and HTTPS) are also widely available. Using an Internet search engine will locate dozens of data collection and analysis tools from which you can choose.

So far, you have seen intelligent test agents drive systems to check functionality, scalability, and performance. It makes sense, then, to have agents record their actions to a central log for later analysis. After all, agents have access to the Internet protocols needed to log their activity to a Web-enabled logging application.

In an intelligent agent environment, collecting results data requires the following considerations.

What Data to Collect

Data collection depends on the test criteria. Proofing the functional criteria will collect data on success rates to perform a group of functions as a transaction. A test proofing scalability criteria collects data on the individual steps taken to show which functions scale well. Proofing performance criteria collects data on the occurrences of errors and exceptional states.

At a minimum, test agents should collect the time, location, and basic information on the task undertaken for each transaction. For example, when proofing functionality of a Web-enabled application, a test agent would log the following result data:

```
Agent       Task         Result  Module          Duration
Stefanie 1  Sign-in      Ok      com.ptt.signin  00:00:00:12
Stefanie 1  Run Report   OK      com.ptt.report  00:00:08:30
Stefanie 1  Send Results OK      com.ptt.send    00:00:00:48
```

For functional testing, the results need to show that each part of the overall test functioned properly, and they also show how long each step took to complete. Some test protocols describe the overall test of a function as a use-case, where the setup parameters, steps to use the function, and expected results are defined. When proofing scalability, the Stefanie agent logs the following result data:

```
Agent    Task               Results Time        Duration
Chris 1  Sign,report,send Ok      14:20:05:08 00:00:09:10
Chris 2  Sign,report,send Ok      14:25:06:02 00:00:06:12
Chris 3  Sign,report,send Ok      14:28:13:01 00:00:08:53
Chris 4  Sign,report,send Ok      14:32:46:03 00:00:05:36
```

Scalability testing helps you learn how quickly the system handles users. The result data shows when each agent began and how long it took to finish all the steps in the overall use-case.

Where to Store the Data

By default, Web-enabled application software packages log results data to the local file system. In many cases, this becomes dead data. Imagine tracking down a remote log file from a Web-enabled application in a grid of networked servers! Retrieving useful data is possible, but it requires much sleuthing. In addition, once the results data is located, analysis of the data can prove to be time consuming.

In my experience, the best place for results data is in a centralized, relational database. Databases—commercial and open source—are widely available, feature inexpensive pricing options, and come with built-in analysis tools. Database choices include fully featured relational systems with the Structured Query Language (SQL) to a flat file database manager that runs on your desktop computer.

Understanding Transparent Failure

As a tester it is important to keep a bit of skepticism in your nature. I am not recommending the *X-Files* level of skepticism, but instead you should keep an eye on the test result for a flawed test. In this case, the test data may be meaningless, or worse, misleading. In a Web-enabled application environment, the following problems may be causing the test to fail.

Network bandwidth is limited. Many tests assume that network bandwidth is unlimited. In reality, however, many networks become saturated with modest levels of agent activity. Consider that if the connection between an agent and the system is over a T1 network connection, the network will handle only 16 concurrent requests if each request transfers 8 Kb of data. Table 3–4 shows how much traffic networks can really handle.

Table 3–4 Network Capacity to Handle Test Agent-Generated Data; Performance Varies Greatly

Results data size	T1 requests	Ethernet requests	T3 requests
1 Kbytes	132	854	3845
2 Kbytes	66	427	1922
4 Kbytes	33	213	961
8 Kbytes	16	106	480

These numbers are calculated by multiplying the result data size by the maximum capacity of bytes per second. A T1 line can transmit 1 million bits per second. A 100 Mbit Ethernet line can transmit 7 million bits per second. A T3 line can transmit 30 million bits per second.

Not enough database connections. Systems in a flapjacks environment use multiple Web application servers to provide a front end to a powerful database server. Database connection pooling and advanced transactional support mitigates the number of active database connections at any given moment. Database connection pooling is defined in the Java Database Connectivity (JDBC) 2.0 specification and is widely supported, including support in Microsoft technologies such as DCOM. However, some database default settings will not enable enough database connections to avoid running out after long periods of use.

Invalid inputs and responses. Web-enabled applications have methods in software objects that accept inputs and provide responses. The easiest way to break a software application is to provide invalid inputs or to provide input data that causes invalid responses. Web-enabled applications are susceptible to the same input and response problems. A good tester looks for invalid data as an indication that the test is failing. A tester also ensures that the error handling works as expected.

Load balancer becomes a single point of failure. Using a load balancer in a system that integrates Web-enabled applications introduces a single point of failure to the system. When the load balancer goes down, so does the entire system. Modern load balancer solutions offer failover to a simultaneously running load balancer.

So far, I have shown technological considerations—checking for functionality and scalability—for testing Web-enabled applications. Next I cover how management styles impact your testing. Then I show how the test results you desire impact the way you test a Web-enabled application.

Management Styles

The feeling I get when launching a new Web-enabled application must be similar to what television executives feel when they launch a new program. It is a thrill to launch a new program, but it is also scary to think of how many people will be impacted if it doesn't work or meet their expectations.

Many business managers have a hard time with the ubiquity and reach of their Web-enabled application. TCP/IP connections over Internets, intranets, and extranets are everywhere and reach everyone with a browser or Web-enabled application software. The stress causes highly charged emotional reactions from management, testers, and developers alike.

I have seen management styles greatly impact how the design and testing of Web-enabled applications will deliver value to the business. Understanding a management style and your style is important to crafting effective designs and tests. Table 3–5 describes management styles, characteristics, and the effect on design and testing for several management types.

Table 3–5 Management Styles and Design and Testing Strategies

Style	Characteristics	Effect
Hierarchical	Strategy is set above and tactics below. Basic belief: "Ours not to reason why, but to do and die."	The senior-most managers in a hierarchy have already made decisions to choose the servers, network equipment, vendors, and location. The design then becomes just a matter of gluing together components provided by the vendor. Intelligent test agent-based test solutions work well as the management hierarchy defines the parameters of the data sought and an acceptable timeframe for delivery. Developers, testers, and IT managers should look for efficiencies by reusing test agent automation previously created or bought for past projects.

Table 3–5 Management Styles and Design and Testing Strategies (continued)

Systemic	Take a problem off into a separate place, develop a solution on their own, and return to the team to implement the solution.	Systemic managers can use design tools and test automation tools themselves, and are happier when they have command of the tools unaided. Test tools enable systemic managers to write test agents that deliver needed data. Training on test automation tools is important before systemic managers are assigned projects. Providing an easy mechanism to receive and archive their test agents afterward is important to develop the companies' asset base.
Entrepreneurial	Want to keep as many business opportunities going at once as possible. Frugal with the company cash.	An entrepreneur finds opportunity by integrating existing resources to solve an unaddressed problem. Design is often a weaving and patching of existing systems. Testing provides proof that a new business method or technology can reach its potential. Tests should focus on delivering proof-points of how the system will work.
Inexperienced	Often fail, downplay, or ignore the business efficiencies possible using technology in their company.	Design is dominated by price/performance comparisons of off-the-shelf solutions. Testing provides business benefits that must be stated in terms of dollars saved or incremental revenue earned. Speak in a business benefits language that is free from technical jargon and grand visions.

The styles in Table 3–5 are presented to encourage you to take a critical look at the style of the manager that will consume your design and your test data and then to recognize your own style. Taking advantage of the style differences can provide you with critical advancement in your position within the business. Ignoring management styles can be perilous. For example, bringing an entrepreneurial list of design improvements and test strategies to a hierarchical manager will likely result in your disappointment.

Consider this real-world example: A test manager at Symantec showed clear signs of being entrepreneurial and was paired with a hierarchical product manager. The test manager recognized his own entrepreneurial style and changed his approach to working with the hierarchical manager. Rather than focusing on upcoming software projects, the test manager showed how existing test automation tools and agents could be reused to save the company money and deliver answers to sales forecasting questions.

Some styles have a tendency to crash into one another. Imagine the entrepreneurial executive working with a systemic test manager. When the executive needs test data, the systemic test manager may not be around—instead working apart from the team on another problem. Understanding management styles and how they mix provides a much better working environment and much more effective tests.

Service Level Agreements

Outsourcing Web-enabled application needs is an everyday occurrence in business today. Businesses buy Internet connectivity and bandwidth from ISPs, server hosting facilities from collocation providers, and application hosting from application service providers (ASPs). Advanced ASPs host Web-enabled applications. Every business depends on outsource firms to provide acceptable levels of service. A common part of a company's security policy is requiring outsource firms to commit to a service level agreement (SLA) that guarantees performance at predefined levels. The SLA asks the service provider to make commitments to respond to problems in a timely manner and to pay a penalty for failures. Table 3–6 shows the usual suspects found in an SLA.

Table 3–6 Service Level Agreement Terms

Goal	Description	How to measure
Uptime	Time the Web-enabled application was able to receive and respond to requests	Hours of uptime for any week's period divided by the number of hours in a week (168 hours). The result is a percentage nearing 100%. For example, if the system is down for 2 hours in a given week, the service achieves 98.80952% uptime ((168–2)/168). Higher is better.
Response time	Time it takes to begin work on a solution	Average time in minutes it takes from when a problem is reported to when a technician begins to work on a solution. The technician must not be a call center person but someone trained to solve a problem.

Table 3–6 Service Level Agreement Terms (continued)

Restoration	Time it takes to solve a problem	Maximum time in minutes it takes from when a problem is reported to when the problem is solved.
Latency	Time it takes for network traffic to reach its destination on the provider's network	The measurement of the slowed network connection from the Internet/intranet to the server device; the average time taken for a packet to reach the destination server.
Maintenance	Frequency of maintenance cycles	Total number of service-provider maintenance cycles in a one-month period.
Transactions	Index of whole request and response times	Total number of request/response pairs handled by the system. The higher the better.
Reports	Statistics about monitoring, conditions, and results	Total number of reports generated during a 30-day cycle.

SLAs actually give a business two guarantees:

- The service provider agrees to criteria for providing good service. Often in real-world environments, problem reports go unresolved because of a disagreement on the terms of service. The SLA describes all the necessary facets of delivering good service.
- The SLA becomes part of the service provider's everyday risk mitigation strategy. Failing to provide good service results in an immediate effect to the service provider's financial results. When the service provider fails to meet the SLA terms, the provider will refund portions of the service fees to the business. Depending on the SLA, even greater infractions from the SLA will typically cause the provider to pay real cash money for outages to the customer.

At this point in the Internet revolution, it should be common sense to have SLAs in place; however, Web-enabled applications add additional requirements to SLAs. Enterprises delivering systems today are integrating several Web-enabled applications into an overall system. For example, consider a corporate portal for employees that integrate company shipping reports from one Web-enabled application and a directory of vendors from a second Web-enabled application. If different providers host the application, how do SLAs apply to the overall system? What's needed is a Web-enabled application Service Level Agreement (WSLA).

The WSLA's goal is to learn which Web-enabled application in an overall system is performing poorly. The WSLA asks each service provider to deliver a means to test the Web-enabled application and a standardized means to retrieve logged data. The test must speak the native protocols to make a request to the Web-enabled application. For example, the company shipping reports to the Web-enabled application may use SOAP to respond with the reports. Testing the Web-enabled application requires the service provider to make a SOAP request with real data to the live Web-enabled application. The response data is checked for validity and the results are logged.

The WSLA defines a standard way to retrieve the logged data remotely and the amount of logged data available at any given time. Depending on the activity levels and actual amounts of logged data stored, the WSLA should require the service provider to store at least the most recent 24 hours of logged data. The business and service provider agree to the format and retrieval mechanism for the logged data. Popular methods of retrieving logged data are to use FTP services, email attachments, and SOAP-based Web-enabled applications.

A WSLA in place means a business has a centralized, easily accessible means to determine what happened at each Web-enabled application when a user encountered a problem using the system.

Of course, live in the shoes of a service provider for just one day and the realities of Web-enabled application technology begin to set in. A service provider has physical facilities, employees, equipment, bandwidth, and billing to contend with every day. Add to that an intelligent test agent mechanism—which is required to deliver WSLAs—and the service provider may not be up to the task.

As today's Internet technologies move forward, we will begin to see Internet computing begin to look more like a grid of interconnected computers. Intelligent test agent technology is perfectly suited for service providers playing in the grid computing space.

Grid Computing and Intelligent Agents

Twenty-five years into the microcomputer revolution and the computer industry too often relies on computer operators hitting the Reset button to solve system problems. While there is nothing quite as perfect as the Reset button for functionality—it always works—there is something wrong with expecting a system operator to know when to press the Reset button. Short of hiring a bunch of expensive systems analysts, how can we be sure a system is operating at peak efficiency? Letting the system itself know how it is functioning and not depending on functional bottlenecks goes a long way.

Wouldn't it be great if we could build and manage a computer infrastructure offering easy setup, good usability, and low-effort maintenance? This is the new infrastructure envisioned by advocates of grid computing technology and self-healing techniques. Internet users have already benefited from the beginnings of grid computing.

Grid computing takes today's Internet environment and makes it modular. Grids expect every datacenter to have the same basic components: storage, processors, switches, and memory. Grids use standard open protocols to enable datacenters to communicate. With standardized communication, datacenters become commodities and are easily replaced and interchanged. If the New York datacenter isn't working well, for example, traffic is handled by the San Jose datacenter. Offloading from a node on a grid of datacenters also lets a datacenter activate systems to perform self-diagnostics and, in many cases, self-healing. Self-healing can mean as little as pressing the Reset button on a server. Advanced self-healing systems understand how the system is being used and take proactive steps to improve system performance and uptime. The protocols to enable grid computing are in their infancy today, and much work must be done.

Much of the work on grid computing has centered around making the pieces of infrastructure—routers, devices, and servers—interoperate with new technologies such as Peer-to-Peer (P2P) technology, like Sun's JXTA (http://www.jxta.org/), and adaptive network technology, like Sun's Jini (http://www.jini.org/). JXTA enables peer systems to create a virtual network where any peer can interact with other peers and resources directly, even when some of the peers and resources are behind firewalls or are on different network transports. Jini network technology is an open architecture that enables developers to create network-centric services—whether implemented in hardware or software—that are highly adaptive to change. Now here comes

Web-enabled applications, which take existing server functions and enable them to be pushed outward and interoperate in loosely coupled, fine-grained systems. Web-enabled applications are a great advancement to take us to grid computing.

The Road to Easy Setup, Use, and Maintenance

It used to be fashionable to criticize Microsoft as a key factor in the computer industry's inability to deliver systems that were easy to set up, use, and maintain. The argument went that Microsoft, as a single company, was unable to meet the world's computer users' needs because those needs are infinitely varied. Microsoft's monopoly control of desktop operating systems and its use of a closed development methodology prevents software engineers with the ability to make positive changes to DOS and Windows from having access to the code and distributing the changes.

Even with public scrutiny in the federal courts and recent moves to open the Windows source code, Microsoft has assumed the role of pariah among the software development community. The best engineers who volunteer their time to improve open source software are not working on Microsoft software.

The most egregious problems in DOS and Windows brought rise to an entire industry of utility software products from third-party software publishers like Network Associates. The everyday user, upon finding a problem in DOS or Windows, could search retail shops for utility software offering a fix. Delivering software through retail distribution systems has its own problems due to the amount of time it takes for products to hit the shelves. The end user waits for utility software publishers to deliver solutions to problems that should never have appeared in the operating system in the first place. And with retail distribution comes product indecision and extra time to market.

Microsoft and other platform providers have grown to depend on utility software publishers to solve problems in operating systems. They described missing parts of the operating system as "third-party opportunities." The computer industry baked this into the process of releasing new operating system technology. Consequently, users were stuck with buggy, less-than-full-featured operating systems weeks or months after the launch of the operating system.

Then something surprising happened: The major platform providers adopted Web-enabled applications to build their next-generation platforms. Most operating systems and software applications have grown to be so large that they are managed like small universes unto themselves. Within

Microsoft Windows are thousands of libraries, utility programs, storage formats, and interfaces. Empires of control formed within the platform providers, where one team may or may not cooperate with other teams. For example, the Microsoft Internet Explorer team depends on deliverables from the multimedia team. So even within a single operating system, there may be "third-party opportunities" where needed functionality is missing to make the operating system complete. The platform providers—Microsoft, Oracle, IBM, and many others—internally chose to adopt Web-enabled application protocols, including Extensible Markup Language (XML); Remote Procedure Call (RPC); SOAP; WSDL; and Universal Description, Discovery, and Integration (UDDI), to increase interoperability among the internal teams.

It is only a coincidence that by adopting Web-enabled application protocols, it becomes much easier for platform providers to open their systems. Exposing APIs required a huge effort to deliver a complete software development kit (SDK). Web-enabled applications describe themselves by using the power of XML. Once an internal API has been created using Web-enabled application protocols, the platform provider needs only choose which APIs to expose to a developer wanting to plug in new features.

With developers widely adopting Web-enabled application protocols to build their software products, it is now possible to create the protocols needed to enable grid computing and self-healing techniques.

Self-Healing Systems

Imagine what a self-healing system might be like. For an enterprise running a human resources portal for its employees, one might find an integrated system that features management services for an employee's pension plan, controls for tax withholding, and a vacation time manager. Figure 3–9 shows how the portal might look from an employee's browser.

Like most enterprise applications, the portal shows data collected from many services. The pension plan features are provided from a PeopleSoft Web-enabled application, the tax withholding comes from Bank of America's payroll Web-enabled application, and the vacation time manager comes from an internal database. An application server coordinates the Web-enabled application requests needed to present a single, unified Web interface for an employee.

In this scenario, imagine what happens when the vacation time database fails. Modern software driving a portal will detect the error and still try to present as much valid data as possible. Figure 3–10 shows what the employee experiences when one of the Web-enabled applications fails.

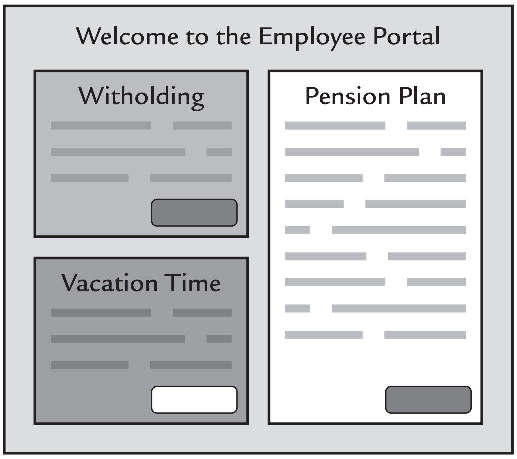


Figure 3–9 Employee's perspective of a human resources portal.

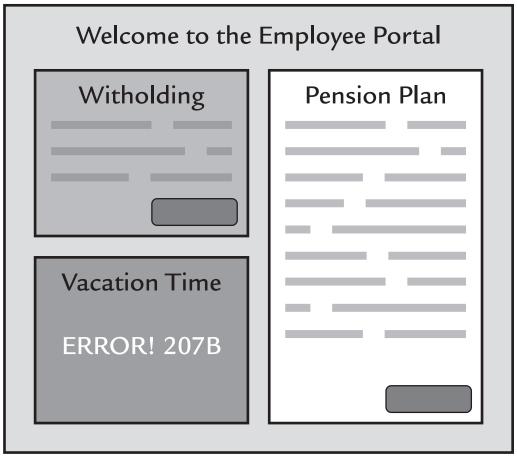


Figure 3–10 Human resources portal in error condition.

The employee has a choice: accept that the function is not working or call someone responsible for the system and ask her to solve the problem. We all have experiences working with service desks to solve problems. At best, the experience is not painful.

Self-healing systems understand the content of the portal page. The system understands when data is missing, inappropriate, or wrong. The key to self-healing grid computing environments is enabling the system to know the difference between good operation and problem operations.

Software developers, IT managers, and QA technicians agree that self-healing systems give better user experiences, provide higher availability, and require less effort to maintain. However, there is usually a debate on responsibility for self-healing technology for a given system. The developer considers self-healing technology to be an extension of unit testing, the IT manager thinks it's part of service-level monitoring, and the QA technician considers it part of an ongoing software regression test.

While self-healing technology may be implemented at the software, test, or operations level, experience shows that the software, such as the application driving the HR portal, should feature controls to manage the application; but the smarts to know when an application is not performing correctly should be built into an external system—a system using intelligent agents to continuously proof a system against performance and scalability criteria. Since intelligent agents have an ability to programmatically respond to the environment they are testing, it makes sense that agents may also be programmed to solve the problems they find.

Understanding Performance and Scalability Criteria

Testing and monitoring a system is an effective technique for building the right-size datacenter, building better Web-enabled applications, and hosting happy users. For a test and monitoring system to be meaningful, we need to understand the criteria for good performance and scalability. The Scalability and Performance Criteria (SPC) defines the needed tests to observe good or bad operation. The SPI is an effective technique to test a system against the SPC criteria. The SPC defines how we know a system is performing well, and the SPI proofs the system against the criteria.

Defining SPC

Criteria puts into words what many of us know in our gut. Experienced Internet users can tell in less than a second when a Web-enabled application is performing badly. Computer use is so widespread today that every computer user has reached minimum criteria for good functionality, scalability, and performance. I find that, on average, Web-enabled application performance comes down to these three points:

1. Are the basic features functioning?

2. How fast is the Web-enabled application performing?
3. How often do requests fail?

By themselves, these three questions are subjective. Imagine a Web-enabled application with basic features that function correctly 95% of the time in an e-commerce application. Would the criteria equally weigh the basic features if a movie listing Web-enabled application functioned 95% of the time? Miss an e-commerce function and the business loses a sale. Miss a movie listing and it's no big deal.

Defining criteria for good Web-enabled application operation depends on many subjective parameters; however, all criteria may be normalized to allow any Web-enabled application to be judged for functionality, scalability, and performance. Following are the ways the SPC normalizes the criteria.

Reward Success

While the systems tested using SPC are a complicated, multifaceted integration of multiple Web-enabled applications, at the end of the day the test is still a measure of how closely the system helps users achieve their goals. SPC rewards the successful achievements of users. The more user goals achieved, the better the operation of the system.

Reward Time Savings and Discourage Slowness

Harkening back to the jingoistic days of phrases like "Time is money," it is evident by now that fast-operating Web-enabled applications are beneficial to users. All user goals are stated with an implication that the users will reach his or her goal in a reasonable amount of time.

Alan Cooper, the father of Visual Basic and a Microsoft fellow, described the term *taction* as to characterize the friction that poorly constructed software interfaces cause to slow the user from reaching goals. Web-enabled applications have taction, too. When an online banking application takes more than 10 seconds to respond to a checking account balance inquiry, the user is slowed in reaching his or her goal. When the same balance inquiry requires that the user access several Web page forms just to get an account balance, that is an example of taction, too.

In an unscientific survey of Web-enabled application users in California and New York, the majority of Web-enabled application users indicated that they would wait 5 seconds or less before clicking the Stop button on their browser.

Users value Web-enabled applications that perform functions in less than 5 seconds and discount Web-enabled applications that take longer. More than 10 seconds is sufficient cause for the user to cancel the operation. SPC rewards fast operation and discourages slow operation.

Discourage Failure

Angry, upset, and unnerved users should be represented proportionally greater than satisfied users. Users get into those states when systems built on Web-enabled applications fail. The failure can come at a protocol level—where the system is unable to handle the request—or at a content level—where the returned data is wrong. For example, a content failure in an employee portal may show another employee's retirement plan information. SPC counts negative points in geometric proportion for operations that fail, compared to operations that succeed.

SPC also discourages failure in proportion to the amount of time the operation took to fail. A content failure that takes less than 1 second to complete is less serious than a content failure that takes 10 seconds. While both end in failure, the faster response wastes less of the user's time. Nothing incenses users more than an application that takes a long time to fail.

Reward Increased Price/Performance

While the cost to buy equipment, bandwidth, and hosting services has dramatically decreased over the past 10 years, these costs become a significant part of the capital budget for a business over time. Systems that handle more users on a given set of equipment are described in a ratio of system costs to the number of users handled. Computer manufacturers coined the term price/performance to describe this ratio. SPC rewards higher price/performance by looking at the number of concurrent users handled at any given time.

Understanding the Jargon

The computer industry is awash with jargon and idioms. In the Web-enabled application-testing world, the terms *concurrent*, *simultaneous*, *synchronous*, *coincident*, and *concomitant* have taken on new lives and specific meanings. Understanding the contrast between concurrent and the other terms is helpful to construct SPC.

- **Concurrent**—Implies parallelism in character or length of time: The system was able to handle 1,000 concurrent users in 1 hour's time.

- **Simultaneous**—Narrowly specifies occurrence of events at the same time: Users organized simultaneous tests of Amazon.com and barnesandnoble.com.
- **Synchronous**—Refers to correspondence of events in time over a short period: The test agents executed a series of synchronous small requests.
- **Coincident**—Applies to events occurring at the same time without implying a relationship: The inventory control system slowing greatly is pretty nearly coincident with the credit card validator slowing.
- **Concomitant**—Refers to coincidence in time of events so clearly related that one seems attendant on the other: The test agents found problems in the database so close in time to the inventory control problem that the problems are concomitant.

While each of these is valid for defining test criteria, SPC emphasizes concurrent testing. Concurrent tests measure the throughput of many requests during a given time period, as opposed to simultaneous or synchronous tests. In a Web-enabled application environment, a request may happen at any moment and take a variable length of time to complete.

SPC in Action

Putting the SPC into action results in a SPI, a metric with many uses, including the following:

- Encourages software developers to optimize code for better user experiences
- Helps IT managers design and deploy the right size data center
- Provides management metrics to judge user satisfaction
- Assists QA technicians to learn whether new software has regressed to include previously solved problems

While the SPC is a fine criterion that is well balanced to test Web-enabled applications, the SPC requires specific inputs depending on the Web-enabled application being tested. Let's look at an example application to see SPC in action. This example features a policy quotation system from a large insurance company in the United States. The insurance company has offices in the western and eastern parts of the United States. They are mandated through regulations to keep records showing the quotes they make to win new business accounts.

The system provides a Web browser front end to the policy quotation system. Insurance agents use a Web browser from their desks to build proposals for business customers. The system also provides market data on the business that the agent many use in the proposal.

The system is run from two geographically distributed datacenters for reasons of redundancy and access speed. The system stores price quotes in a replicate database system accessible from either datacenter. The system uses a Web-enabled application provided by a major research firm to research market data on the business.

In daily operation, the agent signs in to the system and starts a new price quote session or retrieves an existing quote. In the case of new quotes, the system uses the prospective customer's business name and SICS code to retrieve relevant market data on the business. The insurance agent chooses which market data to be included in the price quote file. The agent may optionally attach the market data to the quote as evidence to back up the agent's quote. Figure 3–11 shows the system architecture.

A research firm through a SOAP-based Web-enabled application to the datacenter provides the market data. The agent enters a business SIC code

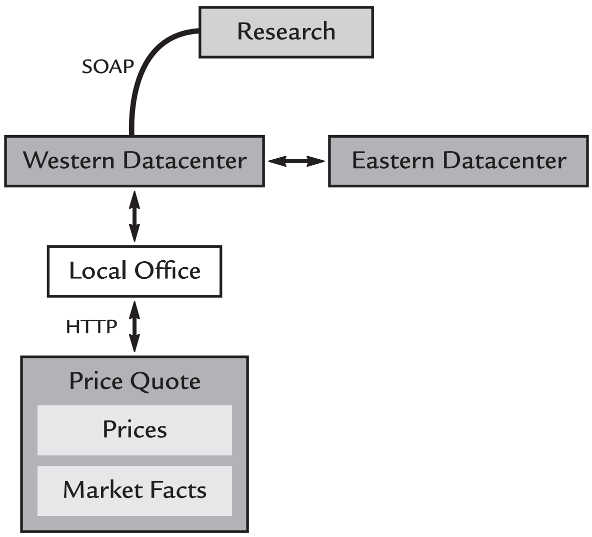


Figure 3–11 Insurance policy quotation system.

and the name of the prospective customer using a Web browser. The local office system makes a SOAP-based Web-enabled application request to the datacenter, which in turn makes a request to the research system. The response contains the research available and the prices available.

The system architecture is typical of modern Web systems, where systems are an integration of various Web-enabled applications. While these systems are prevalent, they are also challenging to IT managers who must answer issues of uptime, datacenter sizing, and problem resolution. Applying SPC shows the nature of this complicated system before rollout and is a metric to measure system performance while in production.

First we look at the SPI results for this system, and then we explore how the SPI was found. Figure 3–12 shows the SPI for the insurance policy quotation system.

The SPI calculation for the policy quotation system breaks down into two parts: Part 1 shows the performance and error measurements recorded when

Part 1: Measurements—15 minutes at 50 concurrent users

Response time	Responses	Weighting	Weighted Responses	Errors	Weighting	Weighted Errors
<1 second	12520	+4	50080	0	0	0
<2 seconds	4050	+3	12150	0	1	0
<3 seconds	3100	+2	6200	16	1	16
<4 seconds	2600	+1	2600	25	2	50
<5 seconds	0	–1	0	43	2	86
<6 seconds	0	–2	0	0	3	0
<7 seconds	0	–3	0	38	3	114
<9 seconds	0	–4	0	45	4	180
<9 seconds	0	–5	0	3	4	12
9 or more	0	–6	0	0	5	0
Total	22270		71030	170		458

Part 2: Analysis

Total weighted responses	71030
out of possible weighted responses	89080
got this close to the goal	79.73731%
However, total weighted errors	458
out of the unweighted response	22270
reduces the overall rating	2.05658%
Giving an overall SPI rating of	77.68074%
Concurrent users tested	50
The SPI is	50.77681

Figure 3–12 The SPI for the insurance example application.

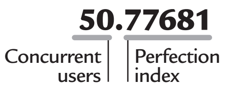


Figure 3–13 Looking closer at the SPI.

50 concurrent users are on the system for 15 minutes. Part 2 shows an analysis of the results to determine the SPI. The SPI is expressed as a number in two parts. Figure 3–13 shows how to interpret the SPI.

The SPI for the policy quotation system is read as "at 50 concurrent users, the system performed to an index value of .77681." A near-perfect rating would be 50.99999, at which level every user is being served in less than 1 second and with no errors. Every response time longer than 1 second and every error eat away at a perfect .99999 index value.

What can be done to improve the SPI rating for this system? The biggest improvements will be to increase the speed at which requests receive responses. Eliminating the responses that took more than 3 seconds will increase the SPI to .67802. Eliminate the errors and the SPI jumps up to .93890.

The full SPI is noted as 50.77681 to indicate the class of system being tested. For example, if the system is hosted on equipment capable of handling a popular public Internet site, we may want to test the system at 5,000 concurrent users. In that case, the SPI would appear as 5000.77681. On the other hand, a small departmental server may need to support only 7 concurrent users, in which case testing the system may result in a 7.77681 SPI rating. In general, the perfection index increases with bigger, faster performing equipment.

SPI is an effective solution in identifying how well a Web-enabled application-based system is able to handle concurrent users at specific levels of load. The SPI is designed to let the IT manager, software developer, and QA technician make what-if decisions to ensure user goals may be met by addressing these questions:

- Do we have the right number of servers to satisfy the expected number of users?
- Are fast response times acceptable while users get error responses?

- Should we spend more money on servers and bandwidth or on software developers?

These questions are the toughest to answer in real-world Web-enabled application environments. They beg for decisions that focus on data center sizes, software performance, and user experience. SPI is an effective tool for making decisions to achieve user goals and for managing development, testing, and maintenance of Web-enabled application-based systems.

Now that you have seen the policy quotation system examined against a SPC and learned the SPI, let's explore how SPC and SPI is crafted for every Web-enabled application-based system.

Modeling a User's Goals

Part 1 of the SPI in Figure 3–12 measures responses by time. For example, in Figure 3–12 the system handled 22,270 requests by responding between 2 and 4 seconds. But what did the system actually respond to? Is a response a transaction, a session, an individual request, or something else? And how does a response translate into a user achieving his or her goals?

SPI measures system performance while an intelligent agent uses the system. The agent performs tasks modeled after a user's goals. To the system, the agent looks like a real user driving a keyboard and mouse to use a browser or software application. To get close to real-world production environments, agents are modeled after several users. The modeling process defines user archetypes, modeling techniques, and agent technology.

User archetypes are prototypical users of a system. In the policy quotation system, these user archetypes might exist:

- **Brigitte**—Branch manager; she usually drinks 4 to 5 coffees a day, which draws comments from her coworkers about her jitteriness. Brigitte needs to retrieve policy quotations from other agents that work in her branch. She reviews the quotations for completeness. She rarely, if ever, writes a new policy herself.
- **Stefanie**—Claims manager. Stefanie and Brigitte seem to encourage each other's coffee-drinking obsession. Stefanie needs to report aggregate policy quotation summaries back to the corporate headquarters. While Brigitte is interested in the

content of the policies, Stefanie monitors the deal flow. Stefanie is less organized than Ed or Brigitte. She normally waits until things pile up before acting.

- **Ed**—Insurance agent. Ed has been writing policies for 30 years and is just 5 years short of retirement age. Ed works slowly with computers. He hardly ever enters incorrect data, but it takes 10 minutes or more to write a new quotation.

User archetypes give insight into the background, work habits, and goals for a prototypical system user. The archetype method is a strong antidote to the usual pitfall in which developers, QA technicians, and IT managers try to solve problems that happen to a broad cross-section of users. Trying to solve problems encountered by the biggest group of users has a defocusing effect in which small problems are fixed but the most significant user problems are left unaddressed.

Implementing intelligent test agents becomes much more straightforward with defined archetypes. For example, an agent based on the Brigitte archetype would constantly check the system for new policy quotations. The agent requests multiple policy quotations at the same time and would bunch the requests into many short bursts. The number of requests would increase every 90 minutes due to Brigitte's many trips to the coffee pot.

Just as they would in a real insurance agency, the archetype users keep business hours and operate concurrently when they are at work. While the Brigitte intelligent test agent is rapidly retrieving policies, other agents based on Stefanie and Ed are on the system, too. The Stefanie agent runs batch-reporting scripts every 90–180 minutes. The Ed agent retrieves market data on new prospective customers and slowly writes a new policy quotation every 45–60 minutes. Ed takes a 5-minute break to smoke a cigarette before starting to write the next quotation.

Agents help to establish near real-world environments to conduct tests. A defined agent is started and allowed to run. Running multiple agents concurrently loads the system. Effective tests develop three or more user archetypes and run multiple copies of each archetypal agent concurrently. For example, the results from Figure 3–12 used 30 Ed agents, 15 Stefanie agents, and 5 Brigitte agents concurrently for a total of 50 agents.

Archetypes model a user's goals by declaring the destination states a user will achieve through use of the system. This may be tricky, as some goals are actually just partial steps taken to achieve a final goal. For example, in the insurance company example, the Ed agent may have a goal of signing in to the system, but this is just a single step to achieving the larger goal of sending a new policy quotation or following up on a prospective customer to accept a proposal. Understanding normal agent states is important to building a valid test.

Test States

Intelligent test agents model the expected use of a system by prototypical users, so it should come as no surprise that a test agent would step through multiple functions to accomplish the archetypal user's goals. The agent is "stateful." While each agent is as varied as the archetypes, all test agents achieve the following states:

- **Setup**—In this state, the agent has assembled a set of initial values and objects needed to communicate with a Web-enabled application. The setup state may also include the agent actually signing in or performing an initialization or identification action on the system.
- **Establishment**—The agent now takes steps to put the host service into a needed state prior to accomplishing an action. The Ed agent signed in and retrieved market research on a new business. These steps were enough to establish the system for the action test state.
- **Action**—The agent requests that the Web-enabled application create new objects, create new data, or modify existing objects. The Web-enabled application executes a complete transaction in the action state—as opposed to the previous steps, which could be rolled back.
- **Analysis**—Time for the agent to learn the results of the request. The agent examines the response data to make sure the response is appropriate for the request.
- **Conditional recurrence**—The agent looks at the response data and chooses a course of action. The agent may go back and achieve the Establishment-Action-Analysis states.

Software test professionals are often eager to understand the test states and build test scripts. While understanding the test states is important, understanding how the test agents flow from one state to another is equally important. The Unified Markup Language (UML) has long been used by developers and is ideal for identifying test agent states.

Using UML and Code Comments to Model Tests

UML began as a way for software developers to describe the actions and states in a software application. UML is ideal for showing the states in a test agent. UML describes a small set of standard notations to indicate flow and state through a system. For example, the UML for the Stefanie agent is shown in Figure 3–14.

UML defines several simple notations to illustrate a program's state, transitions, uses, and functions. Figure 3–14 shows a simple UML diagram describing a use case. The box surrounding the ovals indicates the boundaries of the Stefanie agent. The actor symbol represents the actions the user archetype, as a UML actor, would take to reach the goal. The lines with solid arrows indicate the dependencies between the use cases. The solid arrow means that one use case depends on the previous use case. For example, the agent may not run the report until after signing in to the system.

UML is easy to learn and handy to use, and many tutorials and books are available to help you learn. In a Web-enabled application testing environment, UML is well worth the effort to learn and use.

Now that the behavior of the agent and the agent states are known, we have enough to set up the system, build sample data, code the intelligent test agents, and determine the SPI.

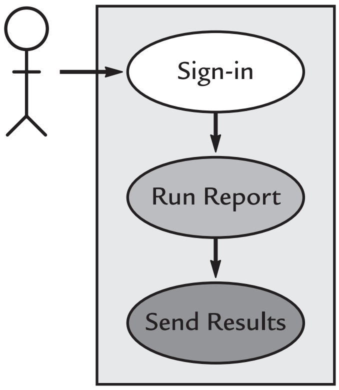


Figure 3–14 UML offers a very simple way to notate a work flow. In this example, the UML diagram shows the Stefanie test agent signing in, running a report, and sending the results.

Putting the Test Together

The agents are written, the server software is installed and configured, the test states are defined, and the user archetypes are written. The test is ready to go! The test begins when the first agent starts. The exact duration and extent of the test depends on the details of the criteria. To get the test results shown in Figure 3–11 for the insurance policy quotation system example, these steps were necessary:

1. Run 50 concurrent agents: 30 Ed agents, 15 Stefanie agents, and 5 Brigitte agents.
2. Run the agents for 60 minutes.
3. Measure the system activity by agent for the last 15 minutes.
4. Run the agents on a mid-range server connected by a 100 Mb Ethernet network to the system host.

The test results in Figure 3–12 show the SPI for this system. Analyzing the results sheds important light on the system. These results also show where to direct the development team to improve the system software.

While the test ran for 60 minutes, only the last 15 minutes of results are significant to this test. In those 15 minutes, the system handled 22,270 agent operations: The Ed agents created new policy quotations and took long breaks in between; the Brigitte agents checked quotations for validity; and the Stefanie agents ran reports for headquarters. SPI is a weighted system that tests the system against the SPC. If all 22,270 agent operations were completed in less than 1 second, the weighted response measurement would have been 89,080 operations. The system missed the performance mark by coming in long. On the average, the system responded in less than 1 second to 56.2% of the requests.

The SPI example data in Figure 3–12 also weights the system for errors. Of the 22,270 agent operations, the system responded with 170 errors. SPI discourages error responses in which the error takes longer than 1 second. In our case, most errors took longer than 5 seconds to complete. Each error is rated according to how long it took to achieve a result. The weighted errors counted for 458 error responses, which is 2.06% of the overall unweighted responses.

Subtracting 2.06% from the error response index from the 79.73% weighted response index gives an SPI of 50.7768. SPI is particularly aimed at the slow moving, poorly performing intranet or Internet systems. It may even appear draconian by nature. However, the 50.77681 SPI is deserved in the Insurance Quotation System since a third (34.4%) of the system's responses to the agent requests took 3 seconds or longer. Adding up the 3 seconds or greater unweighted response times, one can see the system performance.

Summary

I hope you agree that our overarching goal is to develop meaningful test result data that will help us direct our software developers to create better software, help our IT managers build better datacenters, and make for happier users that reach their personal goals. A deeper understanding of software, datacenter, and test agents will benefit any testing and monitoring project and team.

This chapter presented a way to model user goals in a Web-enabled application. We used the user goals to define an SPC. The SPC defines acceptable system performance in our Web-enabled application. From the SPC we developed a metric, called an SPI, to measure system performance against the criteria. This method may be applied to any Web-enabled application.

Chapter 4 introduces the components of a data center needed to test and measure Web-enabled applications. The SPI from this chapter is applied to new datacenter design methods to determine the SPI of an example system.

4
Java Development and Test Automation Tools

The earlier chapters in this part of the book discussed the modern reality of developing and testing Web-enabled applications: it's really, really hard. Building and running Web-enabled applications reliably requires the hard work and cooperation of software developers, QA technicians, and IT managers. Businesses serving every sector of industry, government, and in our personal lives have an incredible opportunity to save money and create new profits by increasing the speed of software development and the reliability of Web-enabled applications.

Over the past 15 years, the computer industry responded to this opportunity with a huge effort to automate software development and testing. We have already gone through three major waves of automation. This leaves the average software developer, QA technician, and IT manager with the daunting challenge to understand and then choose from several competing methodologies and tools.

This chapter describes the three waves of software automation efforts, how to evaluate an automation methodology, and how to assemble a set of software development and test automation tools to deliver improved reliability in Web-enabled applications. Then, Chapter 5 shows a free open source test tool, TestMaker, that I wrote as a test automation framework and utility.

The Three Waves

Since the invention of microcomputers in the 1970s, the computer industry went through three major waves of software development and test automation. These automation efforts parallel the move from desktop applications, to client/server applications, and to today's Web-enabled applications. Table 4–1 describes the three waves.

Table 4–1 Three Waves of Software Development

Wave	Type of application	Development style	Test style
1	Desktop	Event-driven framework surrounds individual procedural functions. The common style is to have a hierarchical set of menus presenting a set of commands.	Test each function against a written functional specification.
2	Client/Server	Structured programming, commands organized into hierarchical menu lists with each client function having its own code. The common style is to combine a common drop-down menu bar with graphical windows contain controls.	Test each function against a written functional specification and test the server for its throughput to serve clients.
3	Web-enabled	Visual integrated development tools facilitate object-oriented design patterns. The common style is to provide multiple paths to accomplishing tasks. (The net effect is that you can't just walk through a set of hierarchical menus and arrive at the same test result anymore.)	Capture/record/playable watches how an application is used and then provides reports comparing how the playback differed from the original recording.

Desktop Application Development and Test Automation

Desktop software development began in the 1980s with the invention of the microprocessor. The early devices were essentially dedicated systems that were able to run a small library of application software that was distributed on floppy disks through retail software stores and mail-order catalogs. The true geeks among us could even download software from the emerging network of bulletin board systems (BBS). The software was written to provide a friendly interface for information workers: Spreadsheet jockeys, business people needing written reports, and game players. The full spectrum of desktop software could pretty well be categorized into spreadsheet, word processor, database, and entertainment categories since desktop computers were rarely networked to other information resources. When a network existed, such as using a slow modem to connect to an information resource like a bulletin board system, there was rarely a link from the desktop application software to the networked information resource. A very few had something approaching ethernet.

Desktop applications used the keyboard, and then later a mouse, to navigate through windows and a drop-down menu. Inside a desktop application software package one would find an event-driven framework surrounding individual procedural functions. The application managed windows and menu events separately from the application logic. The application commands the operating system to open a window, initialize a drop-down menu, and then waits for the operating system to tell the application that the user chose a menu command or moved a window. In the vernacular of the time, the application software responded when the operating system "fired-off an event." The desktop application software that responded to events was procedural in nature.

Procedural software follows a simple one-shot path that goes from initialization to completion in pursuit of a single goal: to answer the user's command. For example, when the user chooses the Print drop-down menu the operating system launches the event handler code in the application to print the current document. The event handler code looks up the current document, formats the document, and sends the formatted data to the printer. The operating system may provide multitasking—for example, to allow the user to create a new document while an existing document is printing—but the event handler for printing only cares about printing the document.

Building desktop application software using an event-driven framework was incredibly easy compared to the object-oriented nightmares a software developer may encounter today. For example, Java Server Pages offers up three object models for software developers to use in a single application: the object model that binds data from the Java application, the object model to dynamically create HTML tag elements that reference action objects on the server, and a third set of objects to model data stored in a relational database. Ugh! In event-driven frameworks, rarely does an event handler need to call a function outside of its own little code base. The event-driven design pattern had a great impact on software development and testing automation efforts.

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. The test utilities link into desktop applications and try each command as though a user were accessing the menu and window commands.

Aside from the name of the drop-down menu command and the text elements appearing in a window, there is little to explain to a test tool what the context is of a command. So most QA technicians testing a desktop application compare the function of all the menus and windows to a written functional specification document. The variation from the document to the performance shows the relative health of a desktop application.

Desktop application frameworks and functional test tools are very popular today. Most integrated development environments come with a New Application Wizard—or similarly named utility—to build the skeleton of a desktop application from its own internally known framework. For example, popular development tools of the time—for example, Microsoft Visual Studio and Sun ONE Studio—ask developers a few questions about the application they intend to build and then write hundreds of lines of code to implement the skeleton of the application. Popular tests tools of the time—for example, Mercury Interactive WinRunner and WinPerl Guido (http://www.winperl.com/)—captures, verifies, and replays user interactions in a desktop application automatically. These test tools automate the way a QA technician identifies defects and ensures that business processes work flawlessly for the user the first time and remain reliable.

Client/Server Development and Test Automation

The original intent for client/server applications was to separate presentation logic from business logic. In an ideal system design, the client was responsible for presenting the user interface, command elements (drop-down menus, buttons, controls), and displayed results information in a set of windows, charts, and dials. The client connected to a server to process functions and the server responded with data. In reality it rarely worked that cleanly. Client/server designs have put functionality in the server and in the client. Client/server today is marked by a common desktop application that communicates to a central server. As a result, the client code can be just as complex as the server code.

Client/server applications are very popular with businesses because their centralized nature enables huge efficiencies in system maintenance, deployment, and development. Client/server application development offers new possibilities for software development and test automation that drove the second wave of tools and methodologies.

In a client/server environment the protocols are cleanly defined so that all the clients use the same protocols to communicate with the server. This facilitates code reuse, information providing from remote data sources, and intelligent client functions became possible. In response, software development automation tools vendors extended their frameworks to provide client-side and server-side frameworks. The client-side frameworks provide the same functionality of desktop application frameworks plus most of the needed communication code to issue commands to the server and the code needed to automatically update a client with new functions received from the server. The server-side frameworks provide code needed to receive and handle requests from multiple clients, and code to connect to databases for data persistence and remote information providers. Additionally, these frameworks need to handle stateful transactions and intermittent network connections. Stateful transactions require multiple steps to accomplish a task. For example, signing in to a Web application so you avoid having to do so on each subsequent task.

Client/server applications are normally transactional in nature and usually several interactions with the user are needed to finish a single request. For example, in a stock trading application the user begins a transaction by identifying themselves to the server, looking up an order code, and then submitting a request to buy a stock. The client-side application communicates the request to the server, and receives and presents the results to the user. The client-side application normally knows something about the transaction—for example, the client-side application will normally store the user identification and a session code such as a cookie value across the user's interaction with the server-based application. Users like it better when the client-side application knows about the transaction because each step in a request can be optimized in the client application. For example, in the previous stock trading example the client application could calculate a stock trade commission locally without having to communicate with the server.

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 test how the application handles slow or intermittent network performance.
- Automated tests are ideal 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.

Client/server applications provide the same challenge to test automation that we first saw in desktop application test automation: there is rarely anything about the communication between the client and server to indicate to a test automation tool the context of the request. It still looks like bits and bytes moving across a network and not stock trading transactions, requests to buy an airline ticket, and research on home mortgages. This becomes a bigger problem for the QA technician testing a client/server application because testing needs to answer these issues:

- Compare the function of all the menu and window commands displayed to the user in the client application to a written functional specification document.
- Determine the impact of increasing numbers of concurrent client requests on server performance.
- Determine when a request from one client impacts the functionality of another client.

Test automation in a client/server application environment is ideal because it greatly extends the productivity of the QA technicians. However, test automation is more of a necessity because client/server applications raise the bar so high.

Client/server application frameworks and functional test tools are hugely popular today. Most enterprise integrated development tools come with rudimentary client/server frameworks built in. Popular tools of the time—for example, Borland JBuilder, IBM Rational Rose, and Compuware OptimalJ—provide frameworks that write the code for client and server components. Popular test tools of the time—for example, Segue SilkTest, Empirix e-Test, Mercury Interactive LoadRunner, and RadView Web FT—provide tool sets to teach the test automation tool about the protocol between client and server, verify interactions, load test the server, and then monitor the system for quality of service.

Web-Enabled Development and Test Automation

Web-enabled applications provided the opportunity for the third wave of software development and test automation. Web-enabled applications build on the experiences of developing, deploying, and maintaining desktop and client/server applications. Web-enabled applications go further in these areas:

- Web-enabled applications are meant to be stateless. HTTP was designed to be stateless. Each request from a Web-enabled application is meant to be atomic and not rely on any previous requests. This has huge advantages for system architecture and datacenter provisioning. When requests are stateless, then any server can respond to the request and any request handler on any server may service the request. (Of course, we will see in the next part of this book that no application is entirely stateless.)
- Web-enabled applications are platform independent. The client application may be written for Windows, Macintosh, Linux, and any other platform that is capable of implementing the command protocol and network connection to the server.
- Web-enabled applications expect the client application to provide presentation rendering and simple scripting capabilities. The client application is usually a browser, however, it may also be a dedicated client application such as a retail cash register, a Windows-based data analysis tool, or an electronic address book in your mobile phone.

Web-enabled applications are very popular with businesses because they 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 datacenter architecture to favor multiple, smaller server boxes, and are ripe to use object-oriented programming techniques.

Object-oriented programming is attractive to software developers because it makes it easy to deliver individual software objects without forcing the developer to worry about the entire system. Each object is responsible for managing and encapsulating its own data. Plus, the application programming interfaces to a software object offer some context for a call to an object. For example, an object named StockTransactions with a method named BuyStock that takes an integer value named Shares and returns a long value named TransactionNumber gives the software developer a good idea of what the object is for and how to call it. Sadly, no equivalent exists to explain to a test automation tool the context of such an object.

The missing context in a Web-enabled application test automation means that software developers and QA technicians must manually script tests for each Web-enabled application. Plus, they need to maintain the test scripts as the application changes. Web-enabled application test automation tools focus on making the script writing and maintenance tasks easier. The popular test automation tools 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 play back 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. The differences may be slower or faster performance times, errors, and incomplete transactions.

Web-enabled application frameworks and functional test tools are a billion-dollar industry. Yet, all the popular tools of the time—for example, in the commercial tools space: Wily Introscope, Mercury Interactive LoadRunner, Segue SilkPerformer, RadView WebLoad, and Empirix eLoad, and the open source tools space: PushToTest TestMaker, The Grinder, and Apache Jmeter—offer the same basic four features: graphical interface, recorder, playback, and report utility. They all expect you to do the work of writing and maintaining test scripts.

Two significant efforts are underway to solve the test script maintenance problem: application modeling tools that generate tests automatically, and the second generation of Web Service standards that define ways to model the workflows of a Web-enabled application in XML.

From Models to Tests

Many software development and test automation tools companies publish modeling languages that enable a software architect and developer to describe a finished software application. For example, Compuware OptimalJ implements the Object Management Group (OMG) Model Driven Architecture (MDA). MDA enables you to visually describe an application and OptimalJ writes Java code for you that implements the application. The MDA description tells OptimalJ all about the context of each function in the application. Since they are translating the model to code already, it is natural that the product is also able to translate the model into a test suite that checks the application for functionality, scalability, and performance.

Another example of a model-driven system is IBM Rational Rose. It implements the Unified Modeling Language (UML), also from the OMG. UML is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. UML lets a software developer note a “blueprint” for construction. IBM Rational’s products can then write a test suite from the same blueprint.

The downside to using a commercial software development and modeling tool is the vendor lock-in that a business suffers over time. If IBM Rational is not able to keep up with your changing needs for its modeling tool, then there is no other tool to adopt without causing a rewrite—and reinvestment—into the new tool.

From Workflows to Tests

Several concurrent efforts are underway to improve software developer productivity of Web-enabled applications on the Java platform. Consider these technologies and tools listed in Table 4–2.

Table 4–2 Making Web-Enabled Application Development More Efficient

Project	Description
Java Server Faces	Defines an architecture and APIs that simplify the creation and maintenance of Java Server application GUIs. Makes Web pages essentially macros that may be glued together into a Web-enabled application. The specification is at: http://web1.jcp.org/en/jsr/detail?id=127
Java Page Flow	A tool set built upon a Struts-based Web application programming model to leverage the power and extensibility of Struts while also eliminating the difficulties and challenges of building Struts-based applications. Java Page Flow includes runtime support for the Web application programming model and tools that enable developers to quickly and easily build applications based on the model. The central concept and construct of Java Page Flow is called page flow. Basically, a page flow is a directory of Web app files that work together to implement a UI feature. Details are at: http://developer.bea.com/articles/Dew.jsp
BPEL4WS	Business Process Execution Language for Web Services (BPEL4WS) is an open standard from the OASIS group that defines common concepts for an XML-based business process execution language, which forms the necessary technical foundation for multiple usage patterns including both the process interface descriptions required for business protocols and executable process models. Details are at: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

Table 4–2 lists just three of the dozens of efforts underway to improve software developer productivity by introducing modeling languages and notation to the software development process. This bodes really well for test automation. With workflow notation delivered from a server comes automatic creation of test scripts to test these Web-enabled applications. Even without these new workflow notation technologies, I believe the time is right for a fourth wave of software development and test automation tools and techniques.

Achieving the Troika—the Fourth Wave

I am in a unique position from most people in that I regularly get to work with software developers, QA technicians, and IT managers in my day job. In my experience, these three groups rarely work as a single unit. Imagine the *troika* they would form when they work together! A troika is a Russian word describing a group of three horses that pull a wagon while standing side-by-side. It is much easier to shoulder a heavy load if one has help. Imagine having three times the effort devoted to building, deploying, and maintaining a Web-enabled application (Figure 4–1). Chapter 1 discussed some of the common problems in businesses today that keep a troika of software developers, QA technicians, and IT managers from forming. With the techniques, methodology, and tools presented in this book, we can precipitate a fourth wave of innovation and productivity in development and test automation. We can achieve a troika!

Figure 4–1 The most efficient and productive way to test Web-enabled applications is to put software developers, QA technicians, and IT managers to work to increase functionality, scalability, and reliability. These three groups working together form the troika!

From my perspective, we live in a unique time. This is the first time that software developers, QA technicians, and IT managers agree on a framework and infrastructure to build distributed applications: Object-oriented applications, running on a routed global network, using a Web infrastructure. But, just consider the complexity built into today's application software!

On the server side, J2EE, .NET, and open-API technology (for example, the Apache Tomcat engine, Struts library, and JBoss application server) gives us a huge range of APIs: from Web pages, database access, and Web Service interoperability, to asynchronous message queues and email messaging, and literally dozens more. Consequently, we can build sophisticated and powerful applications that leverage the many parts of an enterprise infrastructure. With all these APIs and protocols, we ask the questions in Table 4–3.

Each person in the engineering effort asks: How will I test this application? Consider that there is no single client-side API to test everything that J2EE can do on the server side.

Table 4–3 The Most Basic Questions

Role	Question
Software developer	How will I test this new module in the newly architected system?
QA technician	How will I test this system for scalability and functionality?
IT manager	How will I test this system for reliability and availability?

As a software developer, I find that unit testing is very good to ensure that the server-side software components I write do what I say they will. But I also find that some of my components require the correct state before they may be unit tested. For example, a Java Bean that fires off an announcement email message when 100 new orders have been placed needs to have 100 orders entered to be tested! My need is immediate for the following to be able to automate such a test:

1. A framework to write a test
2. Support for the open-standards environment I use
3. Multiple-protocol support to match the server's capabilities, including at a minimum Web protocols (HTTP, HTTPS), Web Service Protocols (SOAP, XML-RPC), and email protocols (SMTP, POP3, and IMAP)
4. A utility that makes testing automatic every time I change my modules

Imagine that I cobble together the technology, code libraries, and utility software to accomplish these four goals. The resulting test automation framework and utility would enable me to rapidly build test scripts to automate the actions needed to get my Java Bean into the correct state. Plus, these test automation scripts naturally extend what I am already doing with a unit test framework like JUnit.

Using this approach, I find that the QA technicians I work with take my test scripts that check for functionality, and run them in the same test environment concurrently to check the system for scalability, concurrency, and regression problems. When a QA technician finds a problem in my code, we get together and look at the test script logs to locate the problem and learn its cause. All I need is a way to construct, run, and analyze these tests to bring together two of the three parts of the Troika: software developers and QA technicians!

In addition, IT managers will regularly take my functional test scripts and run them for long periods of time. The scripts drive the service using the same native protocols that my real end-users would use. The scripts log the proof that the system is working. And the reports make a fine Quality-of-Service report to management and customers. Ha! We have all three pieces of the Troika: software developers, QA technicians, and IT managers all working together!

From these experiences I found that this troika-building approach is within our reach today. We can build an excellent framework and utility for testing J2EE, .NET, and open-API applications from the client side. The next section shows an institution using a Web-enabled application to gain more customers and reduce the costs of doing business.

A Test Automation Lifecycle

While the right test automation tools, techniques, and methodologies will build the Troika mentioned in the previous section of this chapter, automating a test includes many steps. This section presents those steps as a repeatable lifecycle for test automation.

To present the lifecycle, consider a major university in Florida that intends to roll out a new email communication system to its 15,000 students, teaching faculty, and administrative staff. (Chapter 15 presents a case study of this example.) One third of the students live on-campus. The rest commute to the college or take classes from remote satellite campuses. The email system provides access to email accounts through email client software, such as Microsoft Outlook, Mac OS X Mail, and Eudora Mail, and through a browser-based Web interface. The university has already chosen the Web-infrastructure components to serve the email communication application, including use of a commercial email software package. The university has also already chosen three Internet service providers for connectivity to the email system. The university seeks to certify that its datacenter and bandwidth allotments are ready to serve the students, faculty, and administration.

Figure 4–2 shows the lifecycle of development to deployment for the university email communication system. By studying the lifecycle I hope you will see that the lifecycle may be applied to any Web-enabled application test.

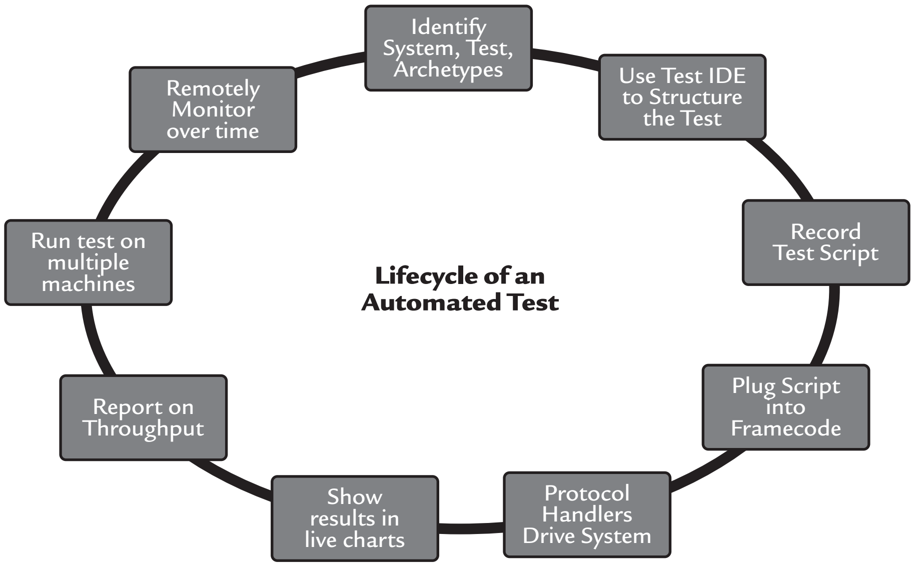

Identify System, Test, Archetypes
Use Test IDE to Structure the Test
Record Test Script
Plug Script into Framecode
Protocol Handlers Drive System
Show results in live charts
Report on Throughput
Run test on multiple machines
Remotely Monitor over time
Lifecycle of an Automated Test

Figure 4–2 The steps to build an automated test of a Web-enabled application.

The test lifecycle is composed of several successive actions. Each action builds on the previous action. The lifecycle tests the email system for scalability, functionality, and performance. By repeating the lifecycle, we additionally test for regression (bugs that were previously solved) as the university changes and upgrades the system. The following describes each step in more detail. Then Chapter 5 shows how you may accomplish these actions using my free open-source test tool called TestMaker.

1. *Identify the test goals, system components, and user archetypes.* The university seeks to certify that its datacenter and network bandwidth are ready to serve the entire student body, faculty, and administration. The system components are a routed network, load balanced group of three servers to provide the Web interface, and four servers running a commercial email package in its cluster mode. The system runs on three networks provided by ISPs, including Florida Telephone, TimeWarner, and Verizon. The university identifies three prototypical users, including Marilyn, a student living on campus; Muriel, a student living off campus; and Betsy, an on-campus instructor.

(Please note: This test and the user archetypes are presented in much more depth in Chapter 15.)

2. *Use an Integrated Test Environment (ITE) to construct the test.* An ITE integrates the functions of a graphical environment, script editor, record and playback utilities, and logging/live-charts capability into one easy-to-use program. While you may certainly use your choice of tools to construct a test, I find that an ITE provides a productivity boost that makes the investment of time and energy to learn the ITE worthwhile. In this part of the lifecycle, the ITE automatically creates a directory to hold the test scenario and test agents that implement the archetype behavior.
3. *Record a test script that drives the browser interface to the application.* The ITE comes with a Recorder function that watches you use a Web browser to accomplish the tasks Betsy normally takes. The Recorder writes the test script that will play back Betsy's activities using the Web interface to check her email mailbox and send messages.
4. *Write a test script that drives the SMTP and POP3 email interfaces to the application.* The ITE comes with script commands to implement the tasks Muriel and Marilyn normally take. The test script plays back Muriel and Marilyn's activities to check and send email messages as though Microsoft Outlook were using SMTP and POP3 email protocols. These scripts are simple functional tests. When run, these tests simply check a single mailbox once and randomly send a single email message.
5. *Plug the test scripts into a test framework to implement an entire test suite.* The ITE provides a test framework to take your functional tests (from steps 3 and 4) and run them as scalability, performance, regression, and functionality tests. Using the framework, you identify the number of virtual users to create and the mix of different archetype behaviors. For example, the university wants to test the email system at 250, 500, and 1,000 concurrent virtual users. For each test, they want 60% of the virtual users to have Muriel's behavior (checking their email mailbox from off-campus), 30% to have Marilyn's behavior (check their email mailbox on campus), and 10% to have Betsy's behavior (faculty sending and receiving large file attachments). The framework gathers your settings from a properties file and automatically stages and executes a series of tests.

6. *Run the tests using the native protocols of the application under test.* The test script drives the university email system using native HTTP, SMTP, and POP3 protocols. To the system it appears a real user is driving the email functions. In reality, the test script language makes special calls to an extensible protocol handler library that implements a wide variety of protocols and is easily extended to support more.
7. *Show the test activity in a set of live charts*. As the test progresses, I find it helpful to see a graphic representation of the live test results in a set of meters and charts. Having a dashboard of test result metrics can show immediate improvements in scalability and performance from changes I make in the infrastructure as the test proceeds. For example, if throughput appears to run slowly I can change router settings to add more bandwidth to the servers and see the result of the change in the dashboard.
8. *Tally the results into a statistical report*. The test generates lots of logged data that are then summarized into one or more reports. For scalability tests, the report shows transactions-per-second, average throughput, an index of errors, and a scalability index. For acceptance tests, the report shows if the application under test passed or failed. For regression tests, the report shows that previously solved bugs are once again in the application. Previous chapters introduced statistical report methods, including Chapter 2 for the *Web Rubric* and Chapter 3 for the *Scalability and Performance Index*.
9. *Run a larger scalability test using multiple test machines*. The test run in step 6 used a single machine to run the test framework. In my experience, an inexpensive 1.4 GHz Intel-compatible system usually runs out of bandwidth and resources at approximately 300 to 500 concurrently running test agents. To achieve the 1,000 concurrent virtual users, the university needs to stage a test on multiple machines. The test environment needs to synchronize the start of the test across multiple machines, manage multiple machines as the test runs, and consolidate and analyze the test results from multiple machines into a single report.
10. *Remotely monitor the application over time*. The university runs the test framework from multiple locations and on multiple networks to understand how well the system is serving users. The

> framework runs the test agent scripts as though Muriel was connecting from her off-campus home, and as though Betsy was on-campus and sending transcripts and faculty reports. Additionally, these tests show how well the ISP's connections to the email service are performing and prove to management the system's quality of service.

The test automation lifecycle defines a set of steps to conduct scalability, functionality, performance, and regression tests on a Web-enabled application. The lifecycle begins again as new builds of software are added to the infrastructure, and when the infrastructure changes (such as when new bandwidth is added or server configurations change). The steps of the lifecycle may be taken using any commercial or open source test tool. The lifecycle is also your checklist against which you may consider which test tools are most appropriate to your Web-enabled application.

Summary

This chapter describes the three major waves of automation since the 1980s. Each wave brought a new set of development and test automation methodologies and tools to market. They are all in use today. This leaves the average software developer, QA technician, and IT manager with the daunting challenge to understand, and then choose from several competing methodologies and tools. This chapter described the three waves of software automation efforts, how to evaluate an automation methodology, and how to assemble a set of software development and test automation tools to deliver improved reliability in Web-enabled applications. In Chapter 5 we learn about TestMaker. It is a free open source test tool that I wrote as a test automation framework and utility to meet your modern needs to build and test scalable and reliable Java Web-enabled applications.

5 Bridging from Methodology to Design

Information systems have reached a level of complexity where almost all new information systems are built, run, and monitored by interdependent groups of people. In response to this, the first part of this book presents the reasons new methodologies for software design, testing, and monitoring are needed. I then presented methodologies and strategies that are designed to leverage the efforts of software developers, QA technicians, and IT managers in a Web-enabled environment. (Chapter 4 called this the troika.) This chapter bridges the discussion of methodologies and strategies to the second part of this book in which I present the technologies, protocols, and tools that are available to you—many times for free—to design, implement, and monitor well-performing and scalable information systems.

In my life, I find it much easier to learn new technology by reading about the fundamentals, learning where the technology may go wrong, and then getting hands-on by using the technology. To get hands-on, I use a common set of tools, including an open source tool named TestMaker that I created and continue to maintain, in my everyday life. This chapter gives you a set of tools that will enable you to rapidly proof scalability and performance of an information system for yourself.

Searching for Tools to Enable the Troika

My search for tools to proof information systems for scalability, performance, and functionality began in 1998. At the time, I was principal architect for the Sun Community Server (SCS) at Sun Microsystems. SCS is a Web-enabled integration server. SCS was one of Sun's first experiments in Web-enabling its messaging, ecommerce, and collaboration server software. The SCS software provides *integrated* access to public and private content through Web pages, email, online conference interfaces, chat, and smart devices. SCS integrates into a corporate database and directory server for single-sign-on functions. For example, using SCS one could easily start a discussion group that enables users to participate through an email or Web interface. When a message is posted to a discussion group, the users that participate through email receive a standard Internet email message containing the posted message. Replies to the email message are posted back to the discussion group in the correct thread order. Web participants would see the email reply messages in a browser-based display of the discussion group. The initial implementations of SCS were well received, with companies like BP, Baker Hughes, and even Sun's own global sales team using SCS to provide a rich, multiprotocol collaboration experience for its employees.

When SCS was ready to be certified for scalability and performance under load, I needed to find a test tool that would be able to check that messages posted by email would appear in the Web interface and vice versa. Here is a check list of features I needed to find in a test tool:

1. A framework to write unit and functional tests. The same tests need to be easily reused for scalability, performance, and regression tests.
2. Support for the open-standards application development environments and a variety of platforms.
3. Multiple-protocol support to match a server's capabilities that may be extended as new protocols are introduced and used. At first I needed HTTP and HTTPS for Web page interfaces, and SMTP, POP3, and IMAP for email functions. Later, I needed SOAP, XML-RPC protocols. Finally, I need a way to add new protocol handlers as they arise.
4. A utility to make recording, editing, and running tests automatically very easy.

5. A way to rapidly understand the test results while the test is running and afterward.
6. A framework to run the functional tests on groups of servers—both local in my lab and remotely around the Internet—in a distributed manner that would log the test results back to a common place.

If that wasn't enough, my budget was already being squeezed downward. I had little, if any, money to spend on a tool.

My search found dozens of possible test tools that would meet my needs. I keep an up-to-date list of commercial and open source test tools at http://www.pushtotest.com/ptt/wiki/TestToolsInfoBook. Compiling the list of test tools showed me that a combination of test tools was necessary to effectively test Web-enabled applications for scalability, performance, and functionality. Unfortunately, I have not yet found a single tool that accomplishes all of these goals. Instead, I found that many of the tools could be made to work with each other. Instead of finding a single tool, I assembled a toolbox. Table 5–1 shows the test tools and libraries I added to my toolbox and why.

Table 5–1 The Contents of Frank's Toolbox

Function needed	**Tool/Library I chose**	**Why I chose it**	**What I gave up**
SOAP requests and responses over HTTP transports	Apache SOAP *http://ws.apache.org/soap/*	Very simple API to learn and use. Supports SOAP RPC and Document Literal encoding. Additionally, the alternatives (Java JAX, found at *http://java.sun.com/webservices/*, and Apache Axis, found at *http://ws.apache.org/axis/*, were not yet available).	Apache SOAP is very slow compared to other SOAP stacks. It does not support WSDL nor SOAP over Java Message Service (JMS) data sources.

Table 5–1 The Contents of Frank's Toolbox (continued)

Scripting language	Jython *http://www.jython.org*	Full featured and object oriented. Runs anywhere Java runs. Full access to Python objects and any Java object.	By choosing one language over the others, I keep the other language users away.
Records browser activity into a Jython script	MaxQ *http://maxq.tigris.org*	Stable code base. Regular maintenence. Easy to customize to write my own test agent scripts.	The design prevents MaxQ from recording browser operations that use SSL and HTTPS.
Bar and Line Charts in SWING controls	JOpenChart *http://jopenchart.sourceforge.net*	Unlike other charting libraries I found, jOpenChart implements an event model to update Swing controls as the underlying data changes.	None that I can think of.
HTTP cookie handling	jCookie *http://jcookie.sourceforge.net/*	Handles IETF RFC 2965 and Netscape style cookies. Nice, light, robust, and well documented.	None that I can think of.
XML data handling	JDOM *http://www.jdom.org*	Very easy and Java-like API to work with XML data. Avoids the complexity of the usual XML parsing libraries.	XML parser libraries are so fast moving that choosing one risks being eventually outdated.
High-performance multidimensional numeric arrays	JNumeric	Good stuff for working with statistical analysis.	None that I can think of.

Table 5–1 The Contents of Frank's Toolbox (continued)

Email protocol support for SMTP, IMAP, and POP3	Java Mail API *http://java.sun.com/products/javamail/*	The standard library for Java developers to work with email protocols.	Flexibility, the Java Mail API is pretty cumbersome to use and integrate.
XML-RPC protocol handler library	Apache XML-RPC *http://ws.apache.org/xmlrpc/*	Simple and easy API to work with hosts using XML-RPC protocols.	None that I can think of.

All of the tools in Table 5–1 have something else in common: they are written entirely in Java and are distributed under free open source licenses. That made it very easy to create a framework and utility application to integrate these libraries and tools. I named the resulting work TestMaker and published it as a free open source test tool for testing Web-enabled applications. TestMaker is available for free download now at http://www.pushtotest.com.

By distributing TestMaker under an open source license, many software developers have provided their input, guidance, and architectural advice over the years. Since its conception, TestMaker has gone through three major revisions. The original software was a simple framework for Java developers to plug in their own test logic. For example, TestMaker 1.0 offered a framework of Java objects that provided a simple way to make HTTP calls to a Web host. Java coders would write the business logic of their test agents as Java objects. This approach proved too brittle. When the target Web host changed in the slightest way, then the Java code in the test objects needed to change too.

Another problem is that the complexity of building tests in Java was a turn-off to many QA technicians and IT managers because it required them to write and maintain object-oriented test agent code in Java. Responding to that, TestMaker 2.0 introduced an XML-based scripting language. Below is an example:

```
<load>
   <script>
      <url action="new" name="myurl"
           host="http://www.pushtotest.com"/>
      <url name="myurl" action="get"
           document="index.html"/>
   </script>
</load>
```

This script creates a URL test object that communicates with the host server to request the index.html page. The TestMaker 2.0 scripting language includes conditionals, loops, and simple expression evaluation. Users were very happy with the introduction of the scripting language but many wanted the language to be more full featured.

Geoff Lane, a leading contributor to TestMaker (and a technical editor of this book), proposed that the XML scripting language in TestMaker 2.0 should be scrapped in favor of using the popular Python scripting language. I already decided that I did not want to build and maintain the TestMaker XML language as a first-class language. (Adding support for regular expressions, full Boolean logic, and all the other trappings of a scripting language gave me nightmares.) Python already had all of these language features.

Then Geoff found Jython, a version of Python written entirely in Java. (Details on Jython are at http://www.jython.org.) Jython compiles scripts written in the Python language into Java byte codes that may be run on any Java Virtual Machine. Test scripts written in Jython can be run on Windows, Unix, Macintosh, and any other system that runs Java. Plus, the test scripts have direct access to all of the Python objects and access to any Java object on the Java classpath! With Jython, any of the libraries listed in Table 5–1 may be called directly from a Jython script. Later in this chapter I provide a complete list of Jython's advantages to people that need to design, code and test information systems. TestMaker 3.0 replaced the XML scripting language and the original test objects with Jython and the Test Object Oriented Library (TOOL) of protocol handlers. TOOL provides a common and extensible programmatic interface to all the need protocols, tools, and libraries.

How to Get TestMaker

TestMaker is available for free download at the PushToTest Web site. Point your favorite browser to http://www.pushtotest.com to download TestMaker. The site features the latest software, archives of previous versions, frequently asked questions about TestMaker, and various technical support services, including email support lists for users and developers.

TestMaker is distributed in a Zip archive file format. Extracting the file contents installs everything that is needed to run TestMaker. TestMaker is a Java application and runs everywhere Java 1.4 or greater runs, including Windows, Linux, Solaris, and Macintosh OS X. Other Java-enabled platforms are capable of running TestMaker but have not been tested.

TestMaker is distributed with both compiled binaries and Java source code. The binaries are ready to run immediately. To build TestMaker from the source code requires the popular free open source Ant build/make utility found at http://jakarta.apache.org/ant and a Java 1.4 or greater compiler.

TestMaker updates appear generally once a month to offer bug fixes and new features. The PushToTest Web site offers a free email announcement service to send email alert messages when new versions become available. PushToTest does not publish or share email addresses.

TestMaker and the Open Source Process

TestMaker introduced me to the world of open source product development. I feel greatly rewarded from the experience. Open source projects are the result of many volunteer software developers who benefit from forming an open source community to solve a real-world problem. When a feature is missing or a bug is found the user has more options than with commercial software. Open source projects shipped with their source code to enable any engineer to solve the problem or add a feature. Depending on the type of open source license, the improvement is contributed back to the project's community for consideration to be incorporated in the project itself.

TestMaker is licensed under an open source license that was modeled after the highly successful Apache Group license. The Apache Group is famous for building and maintaining the Apache Web server and many other open source projects. These projects deliver highly usable, production-ready code that may be used on their own or built into other products. The TestMaker license is found in the license.html file that comes with the TestMaker download. The license allows anyone to download and use TestMaker for free. The license even allows the TestMaker source code to be used to create new products, including commercial software, with no royalty due back to me.

I find that the TestMaker community divides into two groups: users and developers. The PushToTest Web site hosts users@lists.pushtotest.com and dev@lists.pushtotest.com email lists for users and developers to pose questions, share improvements, and generally support each other. The users list is for people wanting to learn more about using TestMaker to test Web-enabled applications. The dev list is for people working on the TestMaker code to make improvements.

Open source projects typically need a principal maintainer to make decisions about the project direction, to choose which improvements are appropriate to the project, and to rally the troops to solve problems and offer solutions. At this writing, I am the principal maintainer for TestMaker. Newcomers to TestMaker submit improvements to be considered for inclusion into TestMaker's code by sending the improvement to the dev@lists.pushtotest.com mailing list. Decisions on submissions are based on the technical proficiency of the code as well as discussion by other contributors. After several improvements have been submitted and accepted, the developer may be added to a list of people privileged with making improvements directly to the TestMaker code. At the time this book was written, five people have commit privileges and 28 engineers actively contribute improvements.

While the tested and released versions of TestMaker are available on the PushToTest Web site, the working copies and interim TestMaker source code is stored in a Concurrent Version System (CVS) server at cvs.pushtotest.com. Anyone with a CVS client may download the latest TestMaker source code from the CVS server. The source code in the CVS server may contain bugs, incomplete features, and experiments; however, the code in CVS is guaranteed to compile. Here are the parameters for downloading the TestMaker working code from CVS:

```
CVSROOT = :pserver:anonymous@cvs.pushtotest.com/var/cvsroot
```

Log into CVS with no password and check out one of these modules:

- tool—retrieves the TOOL library of protocol handlers and test objects
- tm4—the Swing-based TestMaker application

Spending Five Minutes with TestMaker

TestMaker is an easy-to-use utility for building intelligent test agents. This next section shows the steps needed to build a test agent, including:

- Installing TestMaker on a Windows or Linux computer
- Getting to know the TestMaker graphic environment
- Recording a test agent
- Understanding the Jython scripting language and TOOL objects

Installing TestMaker on a Windows or Linux Computer

TestMaker comes as a set of files compressed into a ZIP archive. To install TestMaker, uncompress the archive into a directory. The TestMaker documentation will refer to testmaker_home as the directory into which the TestMaker files are uncompressed.

When upgrading from a previous version of TestMaker, we recommend you make a backup copy of the previous version of the testmaker_home directory. The testmaker_home/settings directory stores your settings, including open window sizes and locations, menu preferences, and other TestMaker options. The testmaker_home/agents directory by default stores the agents created while using TestMaker. Contents from either directory may be copied from previous versions to an upgrade version of TestMaker.

The testmaker_home directory contains several files and directories described in Table 5–2.

Table 5–2 The TestMaker Standard Distribution

Name	Purpose
agents	Example agents to show how TestMaker works through practical examples.
doc	Documentation and additional information on TestMaker, including a reference guide to the TestMaker scripting language and Javadoc documentation for developers wanting to change and improve TestMaker.
source	Source code to build TestMaker.
lib	Compiled Java classes in Java Archive Resource (Jar) packages for TestMaker and all supporting libraries.
util	Utilities to run TestMaker test agents from the Windows, Unix, Linux, and Mac OS X command line.
license.html	The complete license for TestMaker and all the bundled open source projects distributed with TestMaker, including Apache SOAP, Apache Xerces, Jython, Java Mail API, and Java Activation Framework.
testmaker.bat	A launcher script for Windows operating systems.
testmaker.sh	A launcher script for Linux operating systems.
readme.html	Documentation of last-minute changes and additions to TestMaker.

Running TestMaker

TestMaker comes with simple launcher scripts to run on Windows, Linux, and Mac OS X systems:

- Windows users run the testmaker_home/testmaker.bat file. This file launches Java and executes the TestMaker application.
- Unix, Linux, and Mac OS X users run the testmaker_home/testmaker.sh file. This file executes a simple shell script that launches Java and executes the TestMaker application.

When TestMaker starts up you will see setup information describing the system environment, classpath information, version numbers, and other diagnostic information. This information is helpful to determine problems when using TestMaker. Depending on the speed of the computer system, the TestMaker environment should take 30 seconds or less to finish loading.

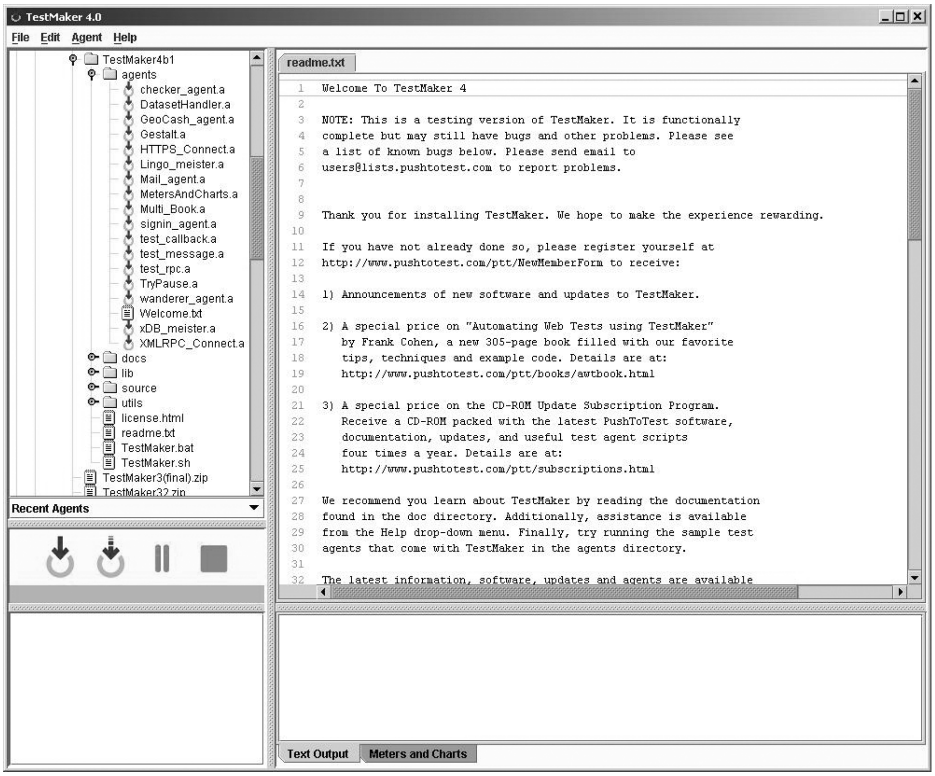
In the event of a problem installing or starting TestMaker, look at the support and frequently asked questions (FAQ) pages on the PushToTest Web site at http://www.pushtotest.com for help.

Getting to Know the TestMaker Graphic Environment

TestMaker 4 is a Swing-based Java application that offers many of the features a software developer, QA technician, and IT manager would expect to find in an integrated development environment. TestMaker comes with a powerful screen editor, output panel, test agent execution controls, visual file system navigator, online help, and a New Agent Wizard.

Figure 5–1 shows the TestMaker graphical environment and many of the elements used commonly to create and run test agents, including the following:

- **Menu bar**—Drop-down menus feature commands to create new agents, print agent scripts, search and replace, and set options for the graphical environment itself.
- **Explorer**—The Explorer is a view into the file system of the local computer. The small, rotating twisty icons expose the files contained in directories. Double-clicking on a file opens the file in the Script editor. The Explorer remembers the directory view for the next time you run TestMaker.



```
Welcome To TestMaker 4

NOTE: This is a testing version of TestMaker. It is functionally
complete but may still have bugs and other problems. Please see
a list of known bugs below. Please send email to
users@lists.pushtotest.com to report problems.


Thank you for installing TestMaker. We hope to make the experience rewarding.

If you have not already done so, please register yourself at
http://www.pushtotest.com/ptt/NewMemberForm to receive:

1) Announcements of new software and updates to TestMaker.

2) A special price on "Automating Web Tests using TestMaker"
   by Frank Cohen, a new 305-page book filled with our favorite
   tips, techniques and example code. Details are at:
   http://www.pushtotest.com/ptt/books/awtbook.html

3) A special price on the CD-ROM Update Subscription Program.
   Receive a CD-ROM packed with the latest PushToTest software,
   documentation, updates, and useful test agent scripts
   four times a year. Details are at:
   http://www.pushtotest.com/ptt/subscriptions.html

We recommend you learn about TestMaker by reading the documentation
found in the doc directory. Additionally, assistance is available
from the Help drop-down menu. Finally, try running the sample test
agents that come with TestMaker in the agents directory.

The latest information, software, updates and agents are available
```

Figure 5–1 TestMaker features an integrated editor, file navigation explorer, output window, and workplace management features.

- **Script editor**—TestMaker comes with a powerful screen editor to write test agent scripts, including clipboard operations to cut, copy, and paste agent code, undo and redo changes, and syntax highlighting. When an agent script encounters an error, the environment highlights the line containing a syntax or other programming error, or the line where the error was found.
- **Output**—A running agent displays progress and statistics in the output window. When more than one agent is executed, the output window uses a tab icon system to switch between agent output displays.
- **Execution control**—Below the Explorer is the Execution control panel that is used to run, pause, and stop test agents. Running tests agents appear in a list below the Execution

control buttons. The left-most Run button will start the currently selected test agent script in the Script Editor window. The Pause and Stop buttons will pause and stop the selected test agent in the list of current running test agents.

Opening and Running Test Agents

Building and running test agents in TestMaker is straightforward and easy. In this section we will run sample test agents and then show how to create a new agent using the Recorder. The screen shots in this section were taken with a freshly installed TestMaker 4.

This tutorial uses the sample test agents in the testmaker_home/agents directory. The agents are fully functional and most connect to the examples.pushtotest.com site, hosted by PushToTest.

To begin, start TestMaker by running the *TestMaker.bat* or *TestMaker.sh* scripts found in testmaker_home/. The TestMaker graphical environment includes features to edit agent scripts, browse file systems for agent files, view output, and control execution of agents. With TestMaker up and running, you may immediately begin using any of the included sample test agents.

TestMaker uses the Explorer to open and manipulate agent scripts. Figure 5–2 shows the first time TestMaker is run, the Explorer shows the highest-level directories. On Windows systems, Explorer shows a list of drives. On Linux systems, Explorer shows the root directory.

Open a sample test agent by using the Explorer to navigate to the testmaker_home/agents directory. Click a twisty icon to reveal the files and directories in a directory. With the agents directory open, double-click the HTTPS_Connect.a sample test agent. Figure 5–3 shows TestMaker with this test agent open.

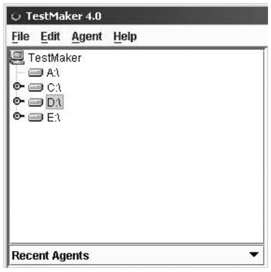
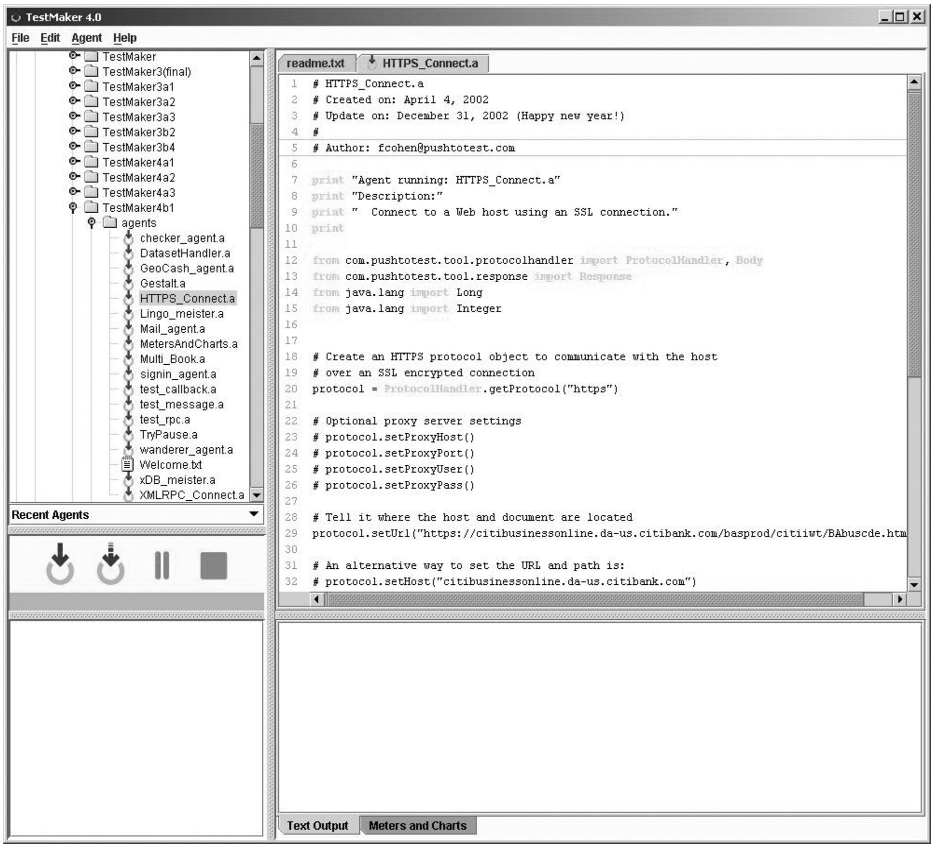


Figure 5–2 The Explorer panel provides easy file navigation to find and open test agent scripts.



```
# HTTPS_Connect.a
# Created on: April 4, 2002
# Update on: December 31, 2002 (Happy new year!)
#
# Author: fcohen@pushtotest.com

print "Agent running: HTTPS_Connect.a"
print "Description:"
print "  Connect to a Web host using an SSL connection."
print

from com.pushtotest.tool.protocolhandler import ProtocolHandler, Body
from com.pushtotest.tool.response import Response
from java.lang import Long
from java.lang import Integer


# Create an HTTPS protocol object to communicate with the host
# over an SSL encrypted connection
protocol = ProtocolHandler.getProtocol("https")

# Optional proxy server settings
# protocol.setProxyHost()
# protocol.setProxyPort()
# protocol.setProxyUser()
# protocol.setProxyPass()

# Tell it where the host and document are located
protocol.setUrl("https://citibusinessonline.da-us.citibank.com/basprod/citiiwt/BAbuscde.htm

# An alternative way to set the URL and path is:
# protocol.setHost("citibusinessonline.da-us.citibank.com")
```

Figure 5–3 TestMaker with the HTTP_Connect.a test agent open.

HTTPS_Connect is a very simple script that makes a single secure HTTPS request to a bank Web site. We will cover the script commands needed to accomplish this agent in the Understanding Scripts section later in this tutorial. For now, please accept that these agents work and that the TestMaker graphical environment offers features to view and execute agents.

To run the agent, click the Run button in the Execution panel in the lower-left portion of the TestMaker window. You may alternatively choose Run from the Agent drop-down menu, or press Control-R.

TestMaker executes agents by creating a new thread for the agent. The running agent appears in the running-agents list below the buttons. Output from

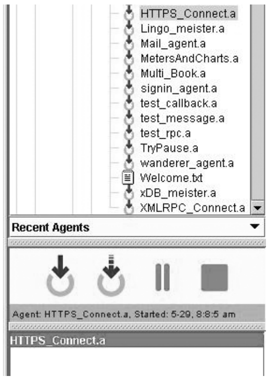


Figure 5–4 The Execution panel is below the Explorer panel and controls running a test agent on the local machine, running a test agent on a distributed set of TestNetwork nodes, pausing a running agent, and stopping a test agent.

the agent appears in the output panel below the script editor window. The output from the HTTPS_Connect agent should look something like this:

```
Agent running: HTTPS_Connect.a

Description:
  Connect to a Web host using an SSL connection.

Received this response from the secure host:
<html><head><meta name="Author" content="gm"><meta name="Pro-
gram" content="BAbuscde.html"><meta name="Description" con-
tent="starting frameset (normal) "><meta name="Version"
content="1.0"><meta name="Last Revision" content="July 12,
2002"><!-- (gm) 7/12/02 -- added util1S.hs --><title>CitiBusi-
ness Online</title><script language="JavaScript" src="js/
util1S.js"></script><script language="JavaScript" src="js/
UtilScreen.js"></script><script language="JavaScript" src="js/
referrer.js"></script><script language="JavaScript" src="js/
BAbuscde.js"></script></head><frameset rows="150,*,130" bor-
der="0" frameborder="no"> <frame name="ltframe" src="BAbus-
cdT.htm" noresize scrolling="no" marginheight="2">      <frame
name="lmframe" src="BAbuscdMI.html"><!-- <FRAME NAME="lmframe"
src="BAbuscdMI.html"> --> <!-- PORTAL version --><!-- <FRAME
NAME="lmframe" src="BAbuscdM.htm"> --> <!-- non-PORTAL version
-->   <frame name="lbframe" src="BAbuscdB.htm" noresize scroll-
ing="no"></frameset></html>

Timing values for this request:

Total: 12568
Data : 6840
Setup: 5528

Agent complete.
```

The HTTPS_Connect agent receives an HTML page over an HTTPS connection secured using SSL encryption—more about security in Chapter 10. The output panel shows the total time it took to complete the request. The value shown is in milliseconds where 1000 milliseconds equal 1 second. The Data time indicates how long the request took before the Web Service began responding with data.

Of course TestMaker also handles more complex calls to Web Services using SOAP and XML-RPC protocols. We will next open a second agent to see TestMaker's graphic environment handling more than one agent at once.

Building Agents with the New Agent Wizard

Next we will see how to create a new test agent using the helpful New Agent Wizard. The Wizard provides four methods to help build test agent scripts. Each method is found in the File ➢ New Agent drop-down menu (Figure 5–5).

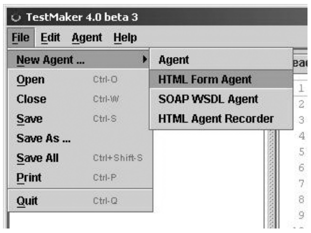
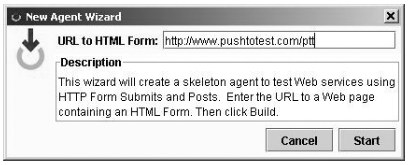


Figure 5–5 The New Agent Wizard provides several helpful utilities to build test agents for you.

The New Wizard is capable of building skeleton agents for testing HTML forms and SOAP-based Web Services. Additionally, the HTML Agent Recorder records your operation of a Web-enabled application through a Web browser and creates a complete test agent script automatically for you. Fist we look at how the HTML Form Agent command works. Then we will use the HTML Agent Recorder to record a Web-enabled application.

The HTML Form Agent command writes a skeleton agent that performs an HTTP `Post` command sending the parameters it finds in an HTML form. You give the New HTML Form Agent a URL to an HTML Web page. The New Wizard retrieves the HTML page and finds any <form> elements. The New Wizard then builds the TestMaker script that will perform an HTTP `Post` command to the servlet referenced by the form. To try the HTML Form Agent, choose File ➢ New Agent ➢ HTML Form Agent. The dialog box in Figure 5–6 appears.



URL to HTML Form: http://www.pushtotest.com/ptt

Description

This wizard will create a skeleton agent to test Web services using HTTP Form Submits and Posts. Enter the URL to a Web page containing an HTML Form. Then click Build.

Figure 5–6 Enter the URL to a page with an HTML form.

Enter http://examples.pushtotest.com/responder/htmlresponder?file=file2.html into the URL entry field. This is a special domain PushToTest hosts that provides several example files and services in support of TestMaker and the sample test agents that come with TestMaker.

Next, click the Start button. In a few seconds the Editor window will display a test agent script that can operate the HTML forms on that Web page. Since this may be your first exposure to the Jython scripting language, I will spend some time explaining the script in detail here. First I will show the entire script and then I will explain the script's individual functions.

```
# Agent name: Try Examples Form
# Created on: July 4, 2003

from com.pushtotest.tool.protocolhandler import \
ProtocolHandler, Header, Body, HTTPProtocol, \
HTTPBody, HTTPHeader

from com.pushtotest.tool.response import Response, \
ResponseLinkConfig, SimpleSearchLink

from java.lang import Exception

# The New Agent Wizard searched the page at
# the given URL for HTML forms.

# First, here is the TestMaker script to
# request the page at the given URL.

http = ProtocolHandler.getProtocol( "http" )
http.setType( HTTPProtocol.POST )
http.setUrl( "http://examples.pushtotest.com/responder/ \
    htmlresponder?file=file2.html" )
response = http.connect()
```

```
print
print response
print

# Form 1

http = ProtocolHandler.getProtocol("http")
http.setUrl( \
 "http://examples.pushtotest.com/responder/htmlresponder")
http.setType(HTTPProtocol.POST)
body = ProtocolHandler.getBody("http")
body.addParameter("firstname", "")
body.addParameter("lastname", "")
body.addParameter("phone", "")
body.addParameter("account", "")
body.addParameter("amount", "")
body.addParameter("firstname", "")
http.setBody(body)
response = http.connect()
print
print response
print

# Form 2

http = ProtocolHandler.getProtocol("http")
http.setUrl( \
 "http://examples.pushtotest.com/responder/htmlresponder")
http.setType(HTTPProtocol.POST)
body = ProtocolHandler.getBody("http")
body.addParameter("firstname", "")
body.addParameter("lastname", "")
body.addParameter("firstname", "")
body.addParameter("Submit", "Update")
body.addParameter("Submit", "Delete")
body.addParameter("firstname", "")
body.addParameter("radio", "value495749")
body.addParameter("comment", "")
body.addParameter("select", "null")
http.setBody(body)
response = http.connect()
print
print response
print
```

Note

The convention used in this book is to break long lines of code using a \ character. You must skip this character and type the code on a single line when entering these example scripts into TestMaker.

Before we look at the details of the Jython script, please take a look at the Web page by pointing your browser to:

```
http://examples.pushtotest.com/responder/htmlresponder?file
=file2.html
```

On this Web page you will find the PushToTest logo, some introductory text, and two forms. Form 1 displays input fields for first and last name, phone number, account number, and amount. The input fields are simple text entry fields. Clicking the Transfer Funds button issues an HTTP `Post` command to the examples.pushtotest.com host. The responder servlet answers the `Post` command by displaying a Web page that includes the Post form values submitted.

This Web page includes a second form. Form 2 displays input fields for first and last name, a text area field for comments, a radio button group for follow-up, and a multiple-choice list for color. Clicking the New, Update, and Delete buttons issues an HTTP `Post` command to the examples.pushtotest.com host. The responder servlet answers the `Post` command by displaying a Web page that includes the Post form values submitted.

Now we will look at the test script in detail to understand how Jython and the TestMaker TOOL work together to implement a simple test.

```
from com.pushtotest.tool.protocolhandler import \
  ProtocolHandler, Header, Body, HTTPProtocol, \
  HTTPBody, HTTPHeader
```

This command, and the two commands that follow it, identifies to Jython the Java objects that the script will use to communicate with the examples.pushtotest.com host. This script will use HTTP protocols to communicate with the host, so the script imports the `HTTPProtocol` object and several others. `HTTPProtocol` provides all the methods necessary to make `GET` and `POST` requests to the host.

For convenience the New Agent Wizard created script commands to get the Web page.

```
http = ProtocolHandler.getProtocol( "http" )
```

This command uses the `ProtocolHandler` object to return a new `HTTP-Protocol` object that the script will later reference using the http variable.

```
http.setUrl( \
 "http://examples.pushtotest.com/responder/htmlresponder \
 ?file=file2.html" )
```

The first use of the http variable is to access the setUrl method of the `HTTPProtocol` object. This tells the `HTTPProtocol` object which URL to use when it connects to the host.

```
response = http.connect()
```

The `connect` method instructs the `HTTPProtocol` object to issue a `GET` request to the host. It returns a new `HTTPResponse` object. The `response` variable now points to the `HTTPResponse` object and may access many included methods to work with the response data. For example, `response.getTotalTime()` returns the total time it took to connect to the host, make the request, and get all of the data back. See the testmaker_home/docs directory or the http://docs.pushtotest.com site for a complete listing of TOOL objects and methods.

```
print
print response
print
```

These commands print the contents of the `HTTPResponse` object to the output window.

The New Agent Wizard searched the page at the given URL for HTML forms. The following code works with the first form.

```
http = ProtocolHandler.getProtocol("http")
http.setUrl( \
 "http://examples.pushtotest.com/responder/htmlresponder")
```

You have already seen these two commands used in the GET request in this script. So I will skip explaining them again.

```
http.setType(HTTPProtocol.POST)
```

By default, the HTTPProtocol object issues an HTTP Get command to the host. Since this is a form, we use the setType method to tell it to make an HTTP Post request to the host.

```
body = ProtocolHandler.getBody("http")
```

HTTP Post commands tell the host to expect to find a payload of data immediately after the HTTP header. TOOL uses an HTTPBody object to set the data payload for the Post command. This command creates a new HTTP-Body object that the body variable points to.

```
body.addParameter("firstname", "")
body.addParameter("lastname", "")
body.addParameter("phone", "")
body.addParameter("account", "")
body.addParameter("amount", "")
body.addParameter("firstname", "")
```

These commands tell the HTTPBody object the parameters to send to the host during the HTTP request. There is one command for each name/value pair from the form in the HTML page we found in the given URL.

```
http.setBody(body)
```

The setBody method identifies the HTTPBody object for the HTTPProtocol object. When the HTTPProtocol object connects to the host, it will use the HTTPBody object defined here.

```
response = http.connect()
```

The connect method tells the HTTPProtocol object to create the request, include the HTTPBody parameters, and make a POST request to the host. The connect method returns a new HTTPResponse object that encapsulates the host's response.

```
print
print response
print
```

These commands print the contents of the `HTTPResponse` object to the output window.

The rest of this script repeats the Jython and TOOL commands to make a request for the second form, so I will not cover these commands here. Chapter 6 provides many more examples of using the `HTTPProtocol` object to work with Web hosts.

In this first example, we saw how TestMaker's New HTML Agent Wizard found two HTML forms on a Web page and automatically wrote a Jython script that used TOOL commands to drive the Web host using native HTTP protocols. To run this script, simply click the Run icon in the Execution pane, or choose Run from the Agent drop-down menu. The results appear in the Output panel.

While this might be interesting to you already, the real power of TestMaker, Jython, and TOOL comes next. The next section of this chapter shows how to use the Recorder to create a test agent that will show us the throughput of the Responder host we connected to in the last example.

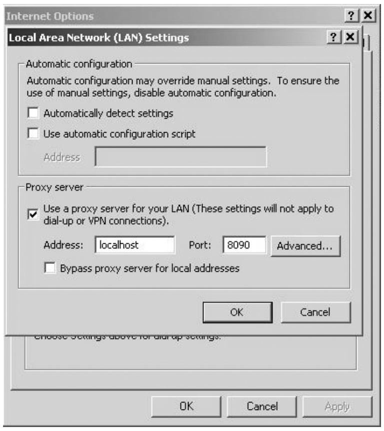
Recording a Test

The Agent Recorder watches you drive a Web-enabled application using your browser and writes a test agent script for you. The resulting test script is fully functional and stages a transactions-per-second test. The script may also be plugged into the XSTest framework—discussed later in this chapter—to conduct scalability, performance, and monitoring tests.

The Recorder is built around a smart proxy server that watches for HTTP traffic between your browser and server. The proxy decodes HTTP `GET` and `POST` commands from the browser and the responses from the server. The proxy then writes the Jython and TOOL script commands necessary to replay your use of the Web-enabled application. This design supports HTTP 1.0 and 1.1 compliant browsers, including the use of JavaScript, plug-ins, ActiveX objects, and Java applets.

The Agent Recorder does not support HTTPS connections. By the time the TestMaker proxy receives the request from the browser, the request is encrypted and TestMaker is not able to decode the body of the HTTP protocol. The only workaround I know at this time is to temporarily host your application with SSL encryption turned off while you record tests.

Before using the Recorder you will need to configure your browser to communicate through the Recorder proxy. By default, TestMaker sets the proxy port to 8090. Your browser preferences or options dialog box usually is



Automatic configuration

Automatic configuration may override manual settings. To ensure the use of manual settings, disable automatic configuration.

☐ Automatically detect settings

☐ Use automatic configuration script

Address

Proxy server

☑ Use a proxy server for your LAN (These settings will not apply to dial-up or VPN connections).

Address: localhost Port: 8090

☐ Bypass proxy server for local addresses

Figure 5–7 Configuring the Recorder proxy server settings in Microsoft Internet Explorer 6.0 for Windows 2000.

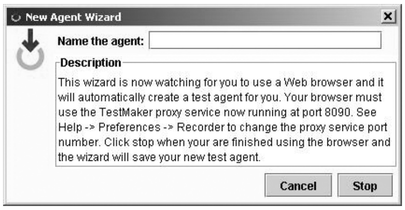
the place to set the used proxy port to be used. Each browser is different at controlling the proxy server. For example, in Microsoft Internet Explorer 6.0 for Windows 2000, the proxy server settings are found by choosing the Internet Options command in the Tools drop-down menu, as illustrated in Figure 5–7. Choose the Connections tab and click the LAN Settings... button. The lower portion of the dialog that appears controls the proxy server settings.

If port 8090 is already in use on your system, then change the TestMaker proxy server to use a different port number. Choose the Preferences command in the Help drop-down menu and then choose the Recorder tab.

With the proxy settings configured, every request from the browser will go through TestMaker's proxy server. This has a side effect in that TestMaker needs to be running for you to use your browser.

Next we will Record a test of the Web-enabled application hosted at examples.pushtotest.com. The application is a simple responder servlet that serves up pages containing random content, responds to HTTP `Post` commands to work with HTML forms, and provides a Web site with HTML links that our test can follow.

To begin, start TestMaker, start and configure your browser, then choose the File ➢ New ➢ HTML Agent Recorder menu command. TestMaker displays a dialog asking for the name of the new test agent, as illustrated in Figure 5–8. Enter MyFirstTest.a as the name of the test agent.


New Agent Wizard

Name the agent:

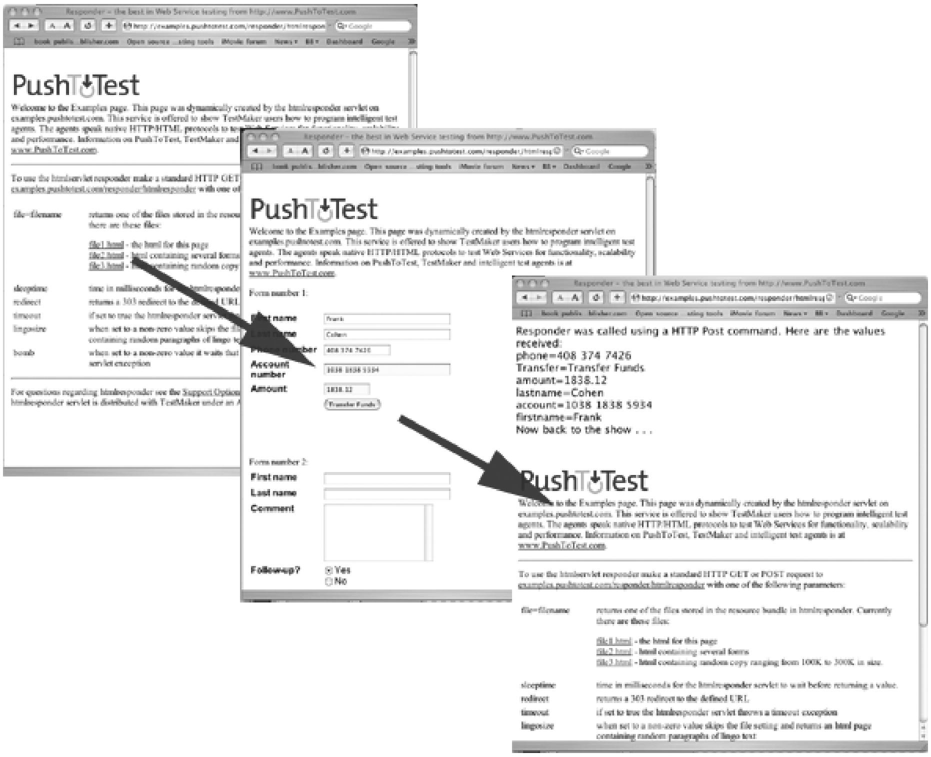
Description

This wizard is now watching for you to use a Web browser and it will automatically create a test agent for you. Your browser must use the TestMaker proxy service now running at port 8090. See Help -> Preferences -> Recorder to change the proxy service port number. Click stop when your are finished using the browser and the wizard will save your new test agent.

Figure 5–8 The New Agent Wizard asks for you to enter a test agent name. The Recorder then watches you use a browser and writes a test script for you automatically.

The test we want to record uses three pages on the examples.push-totest.com site, as illustrated in Figure 5–9. To begin, open your browser and go to this URL:

```
http://examples.pushtotest.com/responder/htmlresponder
```


Welcome to the Examples page. This page was dynamically created by the htmlresponder servlet on examples.pushtotest.com. This service is offered to show TestMaker users how to program intelligent test agents. The agents speak native HTTP/HTML protocols to test Web Services for functionality, scalability and performance. Information on PushToTest, TestMaker and intelligent test agents is at www.PushToTest.com.

Form number 1:

First name	Frank
Last name	Cohen
Phone number	408 374 7426
Account number	1038 1838 5934
Amount	1838.12

Form number 2:

First name	
Last name	
Comment	
Follow-up?	[✓ Yes] No

```
Responder was called using a HTTP Post command. Here are the values
received:
phone=408 374 7426
Transfer=Transfer Funds
amount=1838.12
lastname=Cohen
account=1038 1838 5934
firstname=Frank
Now back to the show . . .
```

To use the htmlservlet responder make a standard HTTP GET or POST request to examples.pushtotest.com/responder/htmlresponder with one of the following parameters:

file=filename	returns one of the files stored in the resource bundle in htmlresponder. Currently there are these files: file1.html - the html for this page; file2.html - html containing several forms; file3.html - html containing random copy ranging from 100K to 300K in size.
sleeptime	time in milliseconds for the htmlresponder servlet to wait before returning a value.
redirect	returns a 303 redirect to the defined URL
timeout	if set to true the htmlresponder servlet throws a timeout exception
lingosize	when set to a non-zero value skips the file setting and returns an html page containing random paragraphs of lingo text
bomb	when set to a non-zero value it waits that [illegible] servlet exception

Figure 5–9 The Web-enabled application presents a page with HTML links, then an HTML form, and finally a dynamically created page showing the form values we submit.

Continue to record the test by clicking on the file2.html link that is about halfway down the page. The browser will display a new Web page that contains two forms. In the first form, enter your first and last name and your phone number. Then enter 1081 as the account number and 75.36 in the amount field. Finally, click the Transfer Funds button. Your browser will show a new Web page that echoes the HTML form information you submitted on the previous page.

Return to TestMaker and click the Stop button. TestMaker will display a standard file selector dialog. Navigate to the directory you wish to store the new agent, enter a file name for the agent, and click the Save button. The new test agent script will appear in the Editor panel in TestMaker. To run the new script, click the run icon in the Execution panel. Alternatively you may choose the Run command in the Agent drop-down menu. The agent will replay the steps you took while using the browser to drive the Web-enabled application on the examples.pushtotest.com domain. However, instead of replaying your action once, the test agent will replay your actions 50 times over. The test agent will then report to you the average transactions-per-second rating for the Web-enabled application.

The Recorder leverages TestMaker's embedded Jython scripting language to efficiently reuse the recorded functional tests for scalability, performance, and monitoring test scenarios. This is possible because TestMaker's embedded Jython scripting language is a full object-oriented language that is capable of building anything you could build with Java and TestMaker's TOOL library provides a uniform and extensible API for protocol handling in test agents. I really like Jython!

Why I Like Jython

For years the Python scripting language has been a popular choice of software developers and QA technicians needing to write quick one-off scripts to perform tests and conduct maintenance tasks. Jython is the Python language implemented entirely in Java. While that might appear to be a nice academic exercise for some computer science student, Jython represents a major move forward for the Java platform. Jython does not just implement Python, it also enables Python scripts to use any Java object. For example, if you are not satisfied with the random number generator object in Python, Jython enables you to use the Java random number generator instead. This flexibility is the cornerstone of TestMaker's ability to build intelligent test agents that implement user archetype behavior and drive Web-enabled applications using the application's native protocols.

Even outside of a test setting, I like Jython. The following is my list of seven reasons why I like Jython.

1. Jython Is Quick

In just a few lines of Jython code, I have a fully working program. For example, here is my first graphical "Hello World" application.

```
import javax.swing as swing
hello = swing.JFrame("Welcome to Jython")
hello.size = (200, 200)
hello.show()
```

I enter this in the TestMaker Editor panel and press the Run icon. Jython creates a java.swing.JFrame object, sets a string value, and displays it on the screen. Using the hello variable I access any and all of the JFrame methods. If I want to see a list of the methods, I can use Jython too:

```
print dir( JFrame )
```

Running this one-line script prints the JFrame object's methods to the TestMaker output window. Jython did not require code compilation, JAR packaging, classpath configuration, or any of the host of other typical Java operations. I find Jython's quickness to be tremendously useful when I prototype and debug test scripts.

2. Data Structures and List Iteration Come for Free

Jython has powerful data structures built in. Plus, I can use Java's collection classes directly from within Jython scripts. For example, suppose I had a list of URLs that I needed to store and later retrieve.

```
urls = { "pageone" : "http://images.pushtotest.com", \
"pagetwo" : "http://examples.pushtotest.com", \
"pagethree" : "http://www.jython.org" }
```

This defines a Jython dictionary object that holds three name/URL pairs. If I need to print a list of the dictionary, I use this command:

```
for i in urls.keys():
  print "The key of ", i, "found the value of ", urls[i]
```

The `for` command in Jython iterates through all the found keys in the URL's dictionary. I could do the same thing using a Java HashMap object but now I have the option: Use Jython's dictionary and list objects or use the Java collections classes. And I can mix the two. (By the way, Sun is working on an enhancement to a future version of Java where for loops incorporate simple iterator creation.)

3. Dynamic Variables with Automatic Typing

Java requires all variables to be created and for their data type to be set before they may be used. In Jython, variables and their types may be dynamically created at any time.

```
def myfunction( x, y ):
    z = x * y
    return z

a = 5
b = 7
print "Result: ",myfunction( a, b )
```

Running this script in TestMaker I would see in the output window:

```
Result: 35
```

Jython creates local variables (z, x, y) and global variables (a, b) dynamically. Jython figures out from the assignment the type of the variables. That is not to say that Jython is an untyped language. Actually, Jython is strongly typed. For example:

```
a = 5 + "1"
```

This command throws an exception because Jython interprets this as "add the integer value 5 to the string value of '1'" and that is illegal in Jython's syntax.

4. Functions Are First Class

Jython functions, methods, modules, and classes are first-class objects. They can be passed and returned from a function call. While Java can do the same using anonymous inner classes or reflection, in Jython, function calls referencing other objects are easier.

```
def scramble( a ):
    return a = a + "ality"

def handleit( mystring, myFunction):
    print

in = "function"
print "Result: ", handleit ( in )
```

Running this script in TestMaker I would see in the output window:

```
Result: functionality
```

Jython's first-class functions, methods, and classes make it easier to write elegant object-oriented code.

5. Java Integration

The above four things I like in Jython are features that come with the standard Python language. Jython goes further by providing very easy access to existing Java code. Jython imports Java packages as though they were Jython modules. When you create an object in Jython, the interpreter creates a first-class Java object for you automatically. These objects are available to your Jython script and also directly accessible from your Java objects.

Jython classes can even inherit from your Java classes. The typical Java inheritance rules work as a Java developer would expect. Jython objects can override parent class methods and parent class methods may be overloaded. In either case Jython chooses the correct version of a method based on the arguments of the call at runtime.

6. Bean Property Introspection

Jython fully supports Java object introspection. Jython automatically calls a Java object's `getAttribute()` method to introspect a method from a Java Bean. This has huge advantages for J2EE developers needing to test Enterprise Beans and Container Managed Persistence. Jython will do the introspection to find the getter and setter methods of an Enterprise Bean to determine what steps an intelligent test agent should take to test an Enterprise Bean for functionality. The same script plugged into XSTest shows how the Enterprise Bean handles the load of multiple concurrent requests.

7. Sun Is Adopting Scripting in Java

A current effort within the Java Community Process (JCP) will add standard support for scripting languages to the Java platform. The JCP is a pseudo open-standards body that works on new features that become part of Java itself. The JCP process identifies needed functionality for Java in a Java Specification Request (JSR.) JSR number 223 describes how to write portable Java classes that can be invoked from a page written in a scripting language, including details on security, resources, and class loader contexts. Details are at http://www.jcp.org/en/jsr/detail?id=223. In my opinion, JSR 223 is a poorly considered approach to adding scripting language support to Java. Read my opinion on JSR 223 at: http://www.pushtotest.com/ptt/wiki/JavaScripting. However, this effort means good things for Jython's longevity and adoption from the larger Java community. That makes Jython even cooler.

I hope these seven reasons why I like Jython encourages you to use Jython and try it on your Java-based projects, even outside of TestMaker. Jython can be downloaded from Jython.org.

Using Jython to Incorporate JUnit

JUnit is a framework for building unit tests of Java classes. It is very popular among software developers and QA technicians because of its simple design and powerful use of objects. Plus, it fits nicely into the "test first" methodology of agile (also known as extreme and XP) programming techniques. Since JUnit is written in Java, it serves as a good example of Jython's ability to integrate test tools and frameworks. In this section, I will introduce JUnit and show how JUnit and any other Java test tool may be used within the TestMaker framework.

JUnit for Repeatable Tests

One of the precepts to agile programming is to *test first*. Literally, a piece of code is not accepted into a source tree unless it includes a test to verify that a function, interface, and implementation does what it is expected to do. This is important because complex interoperating information systems are composed of a myriad of software components and modules. Without a test, changes in one module may break another. JUnit implements a framework for building tests that is automatically run each time the code in a system changes.

Prior to JUnit there were several attempts to create a well understood and widely used framework for testing software modules. Kent Beck's early work to build a unit test framework in Smalltalk led the way to unit test frameworks in Java (http://www.junit.org,) Python (http://pyunit.sourceforge.net/pyunit.html), and .NET (http://sourceforge.net/projects/nunit/). For Kent's original proposal, see http://www.xprogramming.com/testfram.htm.

A unit test framework has a concise set of goals:

1. **Make it familiar to use**. Software developers are going to be writing unit tests in addition to all their other responsibilities. So, the unit test framework has to offer familiar tools that require the absolute minimum amount of work to write unit tests.
2. **Make the tests last a long time**. Testing a new build of software tells us that the software works at this moment in time. But, it does not guarantee that the same module is still working tomorrow, next week, or five years from now. Unit tests need to continue to work over time. It is more than likely that someone other than the original developer will use the unit tests in the future. So, the unit test framework must ensure that the unit tests retain their value over time.
3. **Make it easy to leverage existing tests**. Writing new tests is costly in time and effort. The unit test framework must make it easy to reuse existing unit tests in new tests.

JUnit may be run at any time in the software development lifecycle, but many times JUnit is run by a Java compiler to unit test a newly compiled application. When combined with the popular Ant (http://ant.apache.org/) make utility, unit tests built in the JUnit framework are automatically called by the Ant make script right after the code is compiled. In this environment, unit testing is automatic and transparent. In the software developer's mind, unit tests are just part of the compiler operation. The compiler checks the source code for errors and the unit tests then run the compiled code looking for errors and correct operation.

A JUnit Example

JUnit accomplishes these goals by implementing a basic concept, the `TestCase`. It matches what most software developers do in their everyday lives when building tests. The TestCase object implements three methods: setUp is where a system is configured prior to a test, runTest is a method to run the test, and tearDown performs any needed clean-up tasks. The TestCase implements methods to run a software module and check its response for valid data. Finally, the TestCase implements methods to handle exceptions thrown by the object being tested and to record the test results for later analysis.

To put this into context, consider an object that takes two numbers, multiplies them, and returns the result.

```
public class Utilities
{
    public int multi( int valueone, int valuetwo )
    {
        return valueone * valuetwo;
    }
}
```

The JUnit test to check the multi method in the Utilities class looks like this:

```
public class TestUtilities extends TestCase
{

    public void testMulti()
    {
        int valueone = 24;
        int valuetwo = 2;
        int expectedresult = 48;

        Utilities util = new Utilities();

        int result = util.multi( valueone, valuetwo );

        assertTrue( expectedresult == result );
    }
}
```

Once the test is defined you can run the test by creating an instance of it and calling it the testMulti method.

```
TestCase test = new TestUtilities ("testMulti");
test.run();
```

If all goes well the test will run and the `testMulti` method will return the correct result. If it does not return the expected result then JUnit will throw an exception and show you which test failed.

The JUnit framework provides a group of test runners that enable you to run a suite of tests and collect the results. Here is an example of running two tests in a suite:

```
public static Test suite()
{
    suite.addTest(new TestUtilities("testMulti"));
    suite.addTest(new TestUtilities("testAdd"));
    return suite;
}
```

JUnit and TestMaker

The Jython script language in TestMaker works with any Java object, including JUnit. Working with JUnit in TestMaker means we can leverage the JUnit framework to test Web-enabled applications, EJBs, and Web Services. To add JUnit to TestMaker, follow these steps:

1. Add the JUnit Java Archive Resource (JAR) file to the TestMaker classpath. Do so by editing the *TestMaker_home/TestMaker.sh* file for Unix and Mac OS X systems and the *TestMaker_home/TestMaker.bat* file for Windows systems. For example, for Windows systems add this command:

   ```
   set LOCALCP=%LOCALCP%;%tmlib%\JUnit.jar
   ```

2. Add an import statement to the TestMaker test script that identifies the JUnit classes. For example:

   ```
   from junit.textui import TestRunner
   ```

3. Use the JUnit classes and methods in the test script.

As a practical example of using JUnit in a TestMaker test agent script, consider an example Web-enabled application that transfers money from one bank account to another. The example will use TestMaker's HTTP protocol handler to `POST` the transfer details and JUnit to validate the HTTP response value. Here is the test script in its entirety, followed by a detailed explanation. This example code and all the others in this book are available at http://www.pushtotest.com for immediate download.

```
from junit.framework import TestCase

from com.pushtotest.tool.protocolhandler import \
ProtocolHandler, Body, HTTPProtocol, HTTPBody

from com.pushtotest.tool.response import Response

class TestTransfer( TestCase ):

    def runTest(self):
        self.http = ProtocolHandler.getProtocol("http")
        self.http.setUrl( \
         '''http://examples.pushtotest.com/ \
         responder/htmlresponder''' )
        self.http.setType( HTTPProtocol.POST )

        self.body = ProtocolHandler.getBody( 'http' )
        self.http.setBody( self.body )
        self.body.addParameter('firstname', 'Frank')
        self.body.addParameter('lastname', 'Cohen')
        self.body.addParameter('phone', '374 7426')
        self.body.addParameter('account', '10382')
        self.body.addParameter('amount', '300.00')
        self.body.addParameter('Transfer','Transfer Funds')
        self.response = self.http.connect( 0 )
        self.code = self.response.getResponseCode()

        self.assertEquals(220, \
         self.response.getResponseCode() )

mytest = TestTransfer( "test_transfer" )
mytest.runTest()
print "done"
```

The beauty of this script is that it never leaves the Jython environment. In this one test script we are using Jython objects, TestMaker TOOL protocol handler objects, and JUnit test objects. Here is an explanation of the test script in depth:

```
from junit.framework import TestCase
```

The `Import` command identifies the JUnit object that the test agent script will use. Recall that we modified the TestMaker classpath in the *TestMaker _home/TestMaker.bat* script to include the junit.jar package.

```
from com.pushtotest.tool.protocolhandler import \
ProtocolHandler, Body, HTTPProtocol, HTTPBody

from com.pushtotest.tool.response import Response
```

These Import commands identify the TOOL protocol handler objects that the test agent script will use to communicate with the host and receive the response.

```
class TestTransfer( TestCase ):
```

The test identifies a new object titled `TestTransfer` that will inherit the methods provided in JUnit's `TestCase` class.

```
    def runTest(self):
```

The `TestTransfer` object has a single method `runTest` that performs the actual test. It makes an HTTP Post to the host and receives a response.

```
        self.http = ProtocolHandler.getProtocol("http")
```

The script creates a new `HTTPProtocol` handler object to communicate with the host. The http variable will be the script's reference to the protocol handler.

```
        self.http.setUrl( \
         '''http://examples.pushtotest.com/ \
         responder/htmlresponder''' )
        self.http.setType( HTTPProtocol.POST )
```

The script identifies the URL of the host and a `POST` request to the protocol handler. Now the protocol handler needs the parameters of the transfer function and then it is ready to make the request to the host.

```
        self.body = ProtocolHandler.getBody( 'http' )
        self.http.setBody( self.body )
        self.body.addParameter('firstname', 'Frank')
        self.body.addParameter('lastname', 'Cohen')
        self.body.addParameter('phone', '374 7426')
        self.body.addParameter('account', '10382')
        self.body.addParameter('amount', '300.00')
        self.body.addParameter('Transfer','Transfer Funds')
```

The script creates a new `HTTPBody` object and adds the parameters needed to perform the funds transfer function. In most cases, you would likely use the TestMaker Recorder to create this script. The Recorder automatically determines the parameters by watching you use a browser. See the Recorder section of this chapter for details.

```
        self.response = self.http.connect( 0 )
```

The protocol handler is all ready to make the request to the host. The `connect` method commands the protocol handler to make the request and return a new `response` object. The `response` object returns the response code and provides several handy methods to work with the response data.

```
        self.code = self.response.getResponseCode()

        self.assertEquals(200, \
         self.response.getResponseCode() )
```

With the response code in hand, the script uses JUnit's `assertEquals` method to check that the HTTP response code is actually 200. If the response code is 200 then the script continues to process. If the response code was some other value, JUnit will throw this exception:

```
junit.framework.AssertionFailedError:
expected:<200> but was:<500>
```

In this example, I showed how Jython is easily extended to work with other Java-based test frameworks, including JUnit. Additionally, the combined frameworks of JUnit and TOOL's protocol handlers let JUnit work with Web-enabled applications, something that JUnit offers no specific features for on its own. This is a win-win-win situation!

Summary

This chapter presented a toolbox of test tools and libraries to build intelligent test agents. The practical examples presented in the upcoming chapters in Section II include TestMaker agents to implement real tests for you to get hands-on with the technology presented. While this chapter showed how to install and use TestMaker, including a tutorial on the TestMaker scripting language and a few example test agents, additional details on TestMaker are coming up in subsequent chapters, plus additional information and help from other TestMaker users is available on the PushToTest Web site at http://www.pushtotest.com.

6 Design and Test in HTTP/HTML Environments

By the end of the 1990s the term *The Web* joined the global lexicon, sitting next to other global terms like Coke, Taxi, and Visa. The world's population now knows that when something is "on the Web," they can get it by sitting in front of a browser on their personal computer. Behind the browser is a host of software protocols, routed network connections, and information systems. Those of us who build, test, and maintain things on the Web need to understand the technology that puts things on the Web and a strategy and toolset to test that our resources are really on the Web for our users.

Putting something on the Web requires a combination of technologies. While HTTP is the protocol of the Web, the Hypertext Markup Language (HTML) and XML are the formatting languages. Technologies, such as Java Server Pages (JSP), Servlets, Microsoft ActiveX, Visual Basic, Pearl, Python, Ruby, Macromedia Flash, Graphical Interchange Format (GIF) images, and JavaScript all play their part in enabling software developers to offer a Web resource that is personalized to user needs and preferences. These systems provide desktop-application-like features while leveraging the resources of the servers creating Web pages. This chapter shows how to effectively test Web-enabled applications in HTTP/HTML environments.

The applications we test in this chapter expect that the browser will be fairly "thin" and the server rather "thick"—that is, the browser is only expected to display what the server sends and the server is expected to provide business logic, computational power, interoperability with other systems, and storage. Subsequent chapters present new architectures and test methodologies where the balance between the browser and server blurs.

To start, this chapter demonstrates the basics. We explore the HTTP/HTML environment, including additional content languages like XML and XHTML. Then, we explain how to work with HTML Forms and Cookies. Then we present several strategies for validating a Web-enabled application's responses, including checking a response that is encoded in XML.

The HTTP/HTML Environment

Let us explore the technologies a customer uses in a bank service to transfer funds from one account to another. Most banks today offer Web interfaces. The bank typically exposes a small subset of the functions available to tellers to its customers through a Web browser interface. Behind the scenes, the browser is usually receiving HTML codes over an HTTP connection from a mainframe computer. The mainframe provides the business logic to recognize (or sign-in) a customer and a function to transfer funds from one account to another. The mainframe is likely programmed in COBOL or some other antique programming language. So it should be no wonder that the architecture looks, feels, and operates like an old client/server system—the server commands and the browser obeys.

That is not to say that a Web developer at a bank cannot design and build an elegant Web application. They can and often do; however, the underlying process always works the same. In basic HTTP/HTML architecture, the browser has the following four tasks:

1. It displays HTML codes to the screen.
2. It submits `GET` requests to the server.
3. It submits Form POST requests to the server.
4. It encrypts and decrypts the communication between the browser and server.

Figure 6–1 shows a customer transferring funds from one bank account to another. First, the customer points his or her browser to request the Sign-in page from the bank. The Sign-in page holds a simple form asking for the customer id and password to be entered. The customer clicks a sign-in button and the Transfer page appears in the browser. The Transfer page displays a

Sign-in
Transfer
Confirmation

Figure 6–1 The customer sees three pages in the browser when transferring funds. However, behind the scenes, actually six HTTP transactions take place.

simple form asking for the source account number, the amount to transfer, and the destination account number. The customer enters the values and clicks a transfer button. The customer sees the Confirmation page in the browser.

While the customer only sees three formatted pages during the transaction, twice as many HTTP transactions are actually taking place. Behind the scenes, the browser and server use HTTP protocols to send instructions and/or HTML codes. The browser either gets an HTTP command to display HTML codes or the browser receives a redirection command to make a subsequent server call. In all cases, the browser initiates the request and receives a response from the server.

See Figure 6–2 to observe the browser and server communicating at the HTTP protocol level to better understand the funds transfer transaction. The browser makes requests and the server responds. Each time the communication between browser and server happens this way regardless of how complicated the Web page looks, even when the Web page includes fancy presentation effects, such as animating GIF images, Flash animations, and JavaScript rollover effects.

Figure 6–2 shows each of the `GET` and `POST` commands sent to the server. Each command receives a response, including a response code. For example, the standard response code is 200 to indicate the command was processed successfully and without errors. Table 6–1 describes the steps in the transfer transaction.

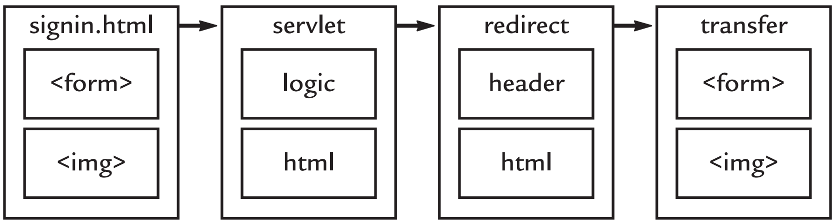


Figure 6–2 In the transfer transaction steps, the browser sends a series of HTTP `GET` and `POST` commands and the server responds with a new Web page each time.

Table 6–1 The Sequence of GET and POST Commands to Process a Bank Transfer

Step	Browser sends command to server	Parameters sent to server	Server's HTTP header response to browser	Server's HTML response to browser
1	`GET`	signin.html	200 OK	HTML containing a sign-in form
2	`POST`	handle_signin servlet	302 Redirect to transfer.html page, also sends cookie identifying this user and session	HTML containing link to the transfer .html page
3	`GET`	transfer.html	200 OK	HTML containing a transfer form
4	`POST`	transfer_funds servlet	302 Redirect to confirmation.html page	HTML containing link to confirmation .html page
5	`GET`	confirmation .html	200 OK	HTML containing the confirmation page

The browser issues `GET` and `POST` commands to the server and receives responses. At a minimum, each `GET` specifies the destination server and the desired Web page, document, or servlet. Generally, the `GET` also sends optional HTTP parameters and HTTP headers to help the server better understand the request. For example, a `GET` may also include a header parameter that passes the current server session information in a parameter called a *Cookie*. Other usual suspects in a `GET` request are an HTTP parameter named Accept that gives the server a list of media types the browser can display. For example, *Accept: text/html, flash/image* tells the server that the browser can display HTML Web pages that contain Flash animations.

A browser's `POST` command looks similar to a `GET` command except for the data being transmitted. `GET` requests send the server a command plus a list of HTTP parameters in the HTTP header. In Step 2 above, the browser makes a `POST` request to the handle_signin servlet and passes the form data containing the user id and password. The `POST` request looks like a `GET` request, however the user id and password are sent in an HTTP body that is a block of data after the request header. This enables `POST` requests to contain an unlimited amount of data.

What Usually Goes Wrong

The HTTP/HTML architecture is so simple! Why should we need to test it at all? What could fail or go wrong? It turns out plenty can and often does go wrong. Here are the most frequent problems found in HTTP/HTML environments:

- Browser noncompliance with standards
- Invalid data
- Session problems

We begin with the biggest problem software developers, QA analysts, and IT managers encounter in HTTP/HTML environments.

Compliance, As in Non, and Browser Caching

A significant reason for the Web's popularity is the absence of a standards body to control the HTTP and HTML specifications. Standards organizations such as the World Wide Web Consortium (http://www.w3c.org) and the Internet Engineering Task Force (http://www.ietf.org) attempt to organize specifications and coordinate efforts from many groups. However, these organizations do not have the power to stop an innovation, or even a poor implementation. To better understand the problem, take a look at WebMonkey's comparison chart of browser compliance found at http://hotwired.lycos.com/webmonkey/reference/browser_chart/index.html. The freewheeling nature of HTTP and HTML enables enthusiastic engineers to freely build and distribute new Web technologies. While this situation benefits the mass of Web users, it causes problems when testing Web-enabled applications.

For example, in the early days of the Web, Internet connections were slow and often failed. Software engineers writing browser software tried to help users by caching portions of the Web page. Clicking the Back button on the browser could then immediately jump to the previously cached Web page instead of having to make the time-consuming request over the slow Internet to the server. This worked well when the majority of Web pages displayed unchanging content that was uniform for all users.

As the Web grew in popularity, caching solutions became important to get documents, movies, graphics, and sound from the server to the browser. By placing caching servers at the edges of the Internet, the delivery speeds increase and bandwidth needs decrease. While caching has its place on the Internet browser, caching is the most significant problem facing Web application developers today. Even though the HTTP 1.1 specification includes caching methods, the most common browsers do not comply with the standard.

Let us examine the caching problem in more depth to see how noncompliance hurts Web applications. Most Web applications dynamically create every Web page to include personalized content for each user. While the HTTP 1.1 specifies caching methods to indicate that a browser should not cache a page's content, these techniques are seldom followed. Microsoft Internet Explorer (MSIE) is the worst offender for showing old data. Clicking the Refresh/Reload button in MSIE does not guarantee your browser connects to the server and actually requests a page.

Imagine the customer using the Sign-in Web page in the previous banking Web application but the browser displays a cached version of the Transfer Web page. The user may not notice the cached version comes up with the previous account number already typed into the form. The server may have stored information into the cached version of the page that makes no sense the second time that it is used. In many cases, browser caching contributes to users not achieving their goals.

Successful strategies to test for and eliminate browser caching focus on defeating the browser's caching mechanism. For example, every response from the server should be tested to see that the HTTP header values contain: **Pragma: NoCache**. (Other HTTP headers include Expires and Last-Modified to help the server know when caching is not wanted.) Although many browsers ignore the No Cache header, using it often helps reduce the cached pages a user encounters. Additionally, the Web page HTML content should include this cache defeating tag in the <head> section:

```
<html>
<head>
  <meta http-equiv="Pragma" content="no-cache">
</head>
<body>
...
</body>
</html>
```

If the browser ignores cache-defeating tags, then your best strategy is to create dynamic Web content that users can use to tell they are viewing cached pages. For example, if every page contains an incrementing simple integer number, then refreshing a page should increment the serial number. A page with the same number indicates the user is viewing a cached page. Additionally, the test can check the date/time values in the HTTP response header.

Invalid Data

Browsers make GET and POST requests to the server using HTTP protocols. The GET request includes a URL, HTTP header information, and a series of name/value pairs. For example, imagine a Web page that offers a list of movies. Each movie name appears as a hyperlink for the user to click. When the user clicks a link, the browser sends a GET request to the server:

```
GET /signin_handler?name=frank&movie=Star%20Wars HTTP/1.0
User-Agent: Mozilla 5.28
Host: examples.pushtotest.com
Accept: text/html, image/gif, image/jpeg, *;
Connection: keep-alive
```

While the HTTP GET command is very lightweight and universally used, it does little to tell the server about the identity of the data. How does the server know that there will be both a *name* and *movie* value? How does it know a valid movie value from an invalid one? Or that the movie value is URL encoded? The browser may construct what it thinks is a perfectly valid GET request, but the server may disagree. Software test strategies for validating data are essential to deploying high-quality HTTP/HTML Web applications.

To catch most problems, you should search for each of the following types of invalid data each time you test a Web-enabled application:

- **Too few or too many parameters**—HTTP/HTML environments have no defined specification of the parameters that will be sent or received. It is up to the developer and HTML designer to agree prior to building the application. Testing a Web-enabled application by sending less than the expected number of parameters will usually turn up broken server logic and security holes.
- **Wrongly ordered data**—Ordering tests for the proper sequence of the occurrence of data. For example, an ordering

test would send a bank account transfer command to a server without first issuing a `GET` command that identifies the back account number in the server session. The test learns how the server handles an out-of-order situation.

- **Boundary data errors**—Range tests the validity of the values. If a name may be no longer than 15 characters, a test determines how the server handles a 17-character-long name.
- **Wrongly formatted data**—There is no schema to define the contents of data in HTTP/HTML environments. Every piece of data is a string of characters. There is also no definition of a character. The HTTP header values in the call may optionally contain a definition of the encoding type (UTF-8), for example.

Let's look at an example of wrongly formatted data in more depth. HTTP/HTML Web applications are particularly vulnerable to invalid data problems because of the nature of HTML. HTML mixes the instructions to lay out a page with the content that appears in the page. Even today popular tools for HTML editing can easily create invalid HTML codes. Special tests must be created to see how the server responds when it receives an invalid HTML form. For example, the following HTML is missing a closing double-quote character in the first input tag:

```
<html>
<body>
<form action="signin_handler">
<input name="signin_name value="Default user">
<input name="password" type="password" value="pass">
</form>
</body>
</html>
```

The server receives a `POST` command that looks like this:

```
POST /signin_handler HTTP/1.1
Referrer: http://examples.pushtotest.com/...
Content-length: 178

signin_name%2Fvalue=&password=pass
```

Note the `signin_name%2Fvalue=` parameter, which is caused by the missing double quote character. Seeing how the server responds to this kind of invalid data is mandatory for a successful test strategy, especially in HTTP/HTML Web applications.

Session Problems

The original design for HTTP/HTML environments was stateless. Each request and response stood alone. Dynamic and personalized Web applications implement state using Cookies, Applets, ActiveX controls, and specially coded URLs. Each time stateful information is introduced, the server needs to record the state data in a session. Intelligent test agents are particularly well suited to test a Web-enabled application for session problems.

Intelligent test agents implement these session tests with ease:

- **Invalid session identities**—Each Web-enabled application formats session identifiers according to its own scheme. For example, the Cookie value for the PushToTest Web site looks like this: 38849198981. Each new user at a unique IP address bumps up the number by 1. A test agent should try valid numbers such as those received from the server. But it should also invent session identifiers to see how the server handles the invalid data.
- **Long sessions**—Each session requires the server to use resources to store session data. The Web-enabled application recycles its resources as sessions end. Test agents may easily push the server resources to maximum by continuing to use the same session information for a long period of time.

As we have seen, many things can and do go wrong in an HTTP/HTML Web application. Constructing and running HTTP test agents is a good technique to find and solve these problems.

Constructing HTTP Test Agents

In this section, we explore constructing HTTP test agent scripts. To get hands-on I will present a complete test script that you can run in TestMaker. Chapter 5 first introduced TestMaker. First I describe the outline of an intelligent test agent and show how the agent script makes requests to the server, validates cookies, sessions, and redirection, and validates the server responses.

The central theme in intelligent test agent technology is to learn a system's scalability, performance, and functional characteristics before customers are exposed to bugs, failures, and scalability problems. Intelligent test agents

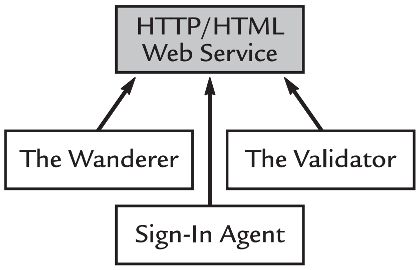


Figure 6–3 Shows an HTTP/HTML Web-enabled application being tested by multiple, concurrently running intelligent test agents.

emulate a user archetype, as in the case of the plodding, slow, and easily distracted Wanderer agent described in the next section. Figure 6–3 shows how the Wanderer is typical of an intelligent test agent that runs concurrently with other agents to simulate a near real-world environment where a server handles many users concurrently. The other concurrently running agents emulate their own user archetypes: The Validator randomly reads and checks the content of Web pages and the Sign-In Agent tries to sign in to a Web-enabled application using a variety of user names and passwords.

The Wanderer is an intelligent test agent that randomly reads pages on a test server hosted by PushToTest, the principal maintainers of TestMaker. The Wanderer initially uses an `HTTPProtocol` object to get a Web page. It then finds hyperlinks on that page and follows a random hyperlink. The Wanderer also keeps track of the time it takes to receive each page. Just for fun the Wanderer pauses after every tenth-loaded Web page and gives an award to the Web page that took the longest time to load.

TestMaker comes with everything needed to create and run the Wanderer, Sign-In, and Validator intelligent test agents. While TestMaker's New Agent Wizard automatically creates intelligent test agents using an easy-to-use graphical user interface, understanding TestMaker's components is important to successfully writing and running your own intelligent test agents (see Figure 6–4).

While Chapter 5 introduced TestMaker, it is important at this point to show how TestMaker's components fit into one another. TestMaker defines the TOOL to provide a common interface to an extensible set of protocol handlers to communicate with servers using HTTP, HTTPS, SOAP, and XML-RPC protocols. TestMaker comes with JDOM, a utility for working with XML data that we will see used by the Validator agent later in this chapter.

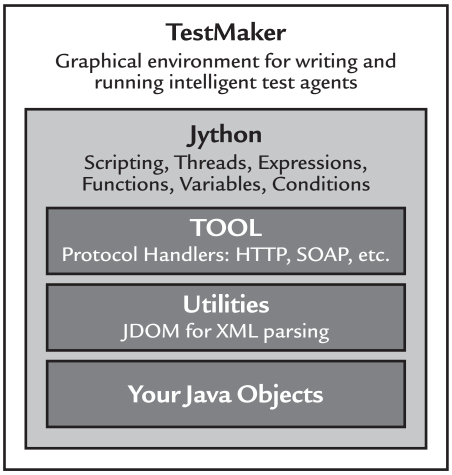

TestMaker
Graphical environment for writing and running intelligent test agents
Jython
Scripting, Threads, Expressions, Functions, Variables, Conditions
TOOL
Protocol Handlers: HTTP, SOAP, etc.
Utilities
JDOM for XML parsing
Your Java Objects

Figure 6–4 An architectural view of the TestMaker environment showing all the components provided to build intelligent test agents.

The Jython scripting language is the glue between your test agent and the TOOL objects. To assist you, TestMaker comes with a Recorder that looks at HTML pages and writes the Jython scripts needed to test an HTTP/HTML Web-enabled application.

TOOL implements an `HTTPProtocol` object you can use for HTTP and HTTPS (secure) protocols, to issue `GET` and `POST` requests, to handle HTTP header parameters (including Cookies), and to search the server response. Figure 6–5 shows an overview of the `HTTPProtocol` object.

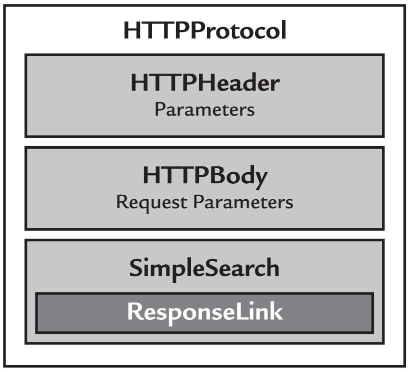


Figure 6–5 TOOL's `HTTPProtocol` object contains objects to connect to an identified host over HTTP and HTTPS protocols, to pass parameters, and to search the results.

The next section demonstrates how the Wanderer agent uses the Jython scripting language to construct an `HTTPProtocol` object that will connect with the server and return a response. While the scripting language is a fully object-oriented language with no test agent specific limitations, it is common practice to separate an intelligent test agent into several parts, including the following:

- Introduction and author credits. This also explains the purpose of the agent.
- Import statements to locate and use TOOL, Java, and Python objects
- Variable definitions
- Function definitions
- Main code
- Post completion analysis and reporting
- Clean-up and finalizers

Hands-On HTTP Communication

Figure 6–3 describes three intelligent test agents concurrently making requests of an HTTP/HTML Web-enabled application. The Wanderer's role is to create load on the Web-enabled application by making requests that cause the Web-enabled application to respond with relatively large blocks of data. The Sign-in and Validator agents' role is to test and validate the Web-enabled application's core functions by requesting functions that require advanced business logic, such as signing in a customer.

The Wanderer uses the scripting language to create and manage HTTP/HTML objects in TOOL. In this section we examine the Wanderer agent to see how Python and TOOL work together. Following is the Wanderer agent in its entirety, followed by a detailed explanation of the Wanderer's components. All of the code presented in this book is also available for download at http://www.pushtotest.com/ptt/thebook.html.

```
# Agent name: wanderer_agent.a
# Created on: May 15, 2002
# Author: fcohen@pushtotest.com

print "Agent running: wanderer_agent.a"
print "Description:"
print "  This agent wanders the examples.pushtotest.com/
responder"

print "  Web site"
print "  Web site finding hyperlinks and following them."
print "  Wanderer also keeps track of the time it takes"
print "  to receive pages."
print "  Every 10 pages wanderer awards the slowest page."
print

# Import tells TestMaker where to find Tool objects

from com.pushtotest.tool.protocolhandler import \
ProtocolHandler, Header, Body, HTTPProtocol, \
HTTPBody, HTTPHeader

from com.pushtotest.tool.response import Response, \
ResponseLinkConfig, SimpleSearchLink

# Import useful Python and Java libraries
from urlparse import urlparse
from java.util import Random

# Global variable definitions

next_url = "http://examples.pushtotest.com/responder"

host = ""        # Holds the decoded host name from a URL
doc = ""         # and the document name from the URL
params = ""      # and the parameters of the call

f1 = '<a href="http://' # Used to search for hyperlinks
f2 = '">'

worsttime = 0    # Tracks the page that took the longest
worstcount = 0
worstname = ""

r = Random()     # A basic random number generator

# hostdoc_decoder: Decodes a URL into the host name

def hostdoc_decoder( theurl ):

    global host, doc, params, next_url, last_good_url
    # urlparse is a handy library function that
    # returns a tupple containing
    # the various parts of a URL, including host,
    # document, parameters, etc.

    parsed_tup = urlparse( next_url )

    # Validate the parsed URL, if it is invalid
    # return with host = null
    # which will signal that another URL is needed

    if ( len( parsed_tup[1] ) == 0 ) :
        host=""
        return

    host = parsed_tup[1]
    doc = parsed_tup[2]
    params = parsed_tup[4]

#    print "host=",host," doc=",doc," params=",params

# Main body of agent

print "Setting-up to make first request."

# Create the needed objects to communicate with the host

httphandler = HTTPProtocol()

# Define a ResponseLink object to search for an <a href> tag

responselink = ResponseLinkConfig()
responselink.setParameter( 'beginsearch', f1 )
responselink.setParameter( 'endsearch', f2 )

# In the TOOL object hierarchy the search parameter
# definition is in a separate object so that a
# single response may have multiple search patterns

search = SimpleSearchLink()
search.init( responselink )

# Find n documents

print "Requesting document: ", doc
```

```
while 1:
    hostdoc_decoder( next_url )

    if host=="":

        # The host we picked isn't valid so
        # raise an exception and end

        raise Spider_Error( "Giving up!" )

    httphandler.setHost( host )
    if params == "":
        httphandler.setPath( doc )
    else:
        httphandler.setPath( doc + "?" + params )

    # Request the document from the host
    response = httphandler.connect()

    # Find the next document URL in the body of the response
    found = search.handle( response )

    # How many found items in the list
    foundcount = found.getParameterValue \
    ("simplesearch.foundcount")

    if ( foundcount == 0 ):
        raise Spider_Error( "No document URLs found." )

    # Pick a URL to load the next document
    foundlist = found.getParameterValues \
    ("simplesearch.founditems")
    doc = foundlist.get( r.nextInt( foundcount ) )

    # Remember the previous host just in case we need to
    # do some backtracking
    last_good_url = next_url

    # Next trim the <a href= and > tags to find the hyperlink

    next_url = "http://" + doc[ len(f1) : ( len(doc) \
    - len(f2) ) ]

    print "links: ",foundcount.toString()," \

    choosing:",next_url
```

```
        print "doc =",doc
        print

        # Time for an award to the page that had the worst time?
        if response.getTotalTime() > worsttime:
            worsttime = response.getTotalTime()
            worstname = last_good_url

        worstcount = worstcount + 1

        if worstcount > 10:
            print "===============Award time================="
            print "The award goes to: ", worstname
            print "which took ",worsttime," in milliseconds \
            to complete."
            print
            worstcount=0

print "Agent finished."
```

The Wanderer makes requests directly to the examples.pushtotest.com server. PushToTest hosts this server, the principal maintainers of TestMaker. Next, we explore the individual parts that make up the Wanderer.

TestMaker bundles Jython, which is the Python language implemented entirely in Java. While it is not necessary to learn Python to use TestMaker, a basic understanding of the language is helpful. TestMaker includes a New Agent Wizard to write and manipulate test agents to help you with the Python language. For help in learning Python, Jython, and TestMaker, see http://docs.pushtotest.com for a list of books and Web resources.

In Jython every Python object is a first-class Java object that may be instantiated, manipulated, called, and destroyed just like any Java object. Jython has the added advantage of being able to work with any Java object directly from the scripting language. The `import` command tells Jython where to find the Python and Java classes that will be used in the agent's script. The format to use a Java object in Jython is:

```
from package import object
```

The import statement makes the `ProtocolHandler`, `HTTPProtocol`, `HTTPBody`, and `HTTPHeader` objects accessible from within a Jython script.

```
# Import tells TestMaker where to find Tool objects
from com.pushtotest.tool.protocolhandler import ProtocolHan-
```

```
dler, Header, Body, HTTPProtocol, HTTPBody, HTTPHeader
from com.pushtotest.tool.response import Response, Response-
LinkConfig, SimpleSearchLink

# Import useful Python and Java libraries
from urlparse import urlparse
from java.util import Random
```

These import statements tell Jython where to find protocol handling objects in Tool and Java objects, such as the urlparse and Random objects. urlparse is a utility object that takes a URL and breaks it down into host, port number, and document parameters. Random is a simple random number generator built into Java. Next we create variables for use later in the agent.

```
next_url = "http://examples.pushtotest.com/responder"

host = ""        # Holds the decoded host name from a URL
doc = ""         # and the document name from the URL
params = ""      # and the parameters of the call

f1 = '<a href="http://' # Used to search for hyperlinks
f2 = '">'

worsttime = 0   # Keeps track of the page that took the longest
worstcount = 0
worstname = ""
```

These commands create variables used in the agent's script. Java developers will notice that Python is a dynamically typed language—in Java every variable is defined by its type before it is used while Python establishes variable types by the type of data being assigned. In Python it is not necessary to identify next_url as a `String` object. Python sees the assignment of "http://examples.pushtotest.com/responder" as a string and automatically identifies next_url as a `String` object variable.

```
r = Random()     # A basic random number generator
```

In the previous code listing we found how simple `String` objects were created. Here we see our first complex object created. The `r` variable will now point to a new instance of the Java Random class.

```
def hostdoc_decoder( theurl ):

    global host, doc, params, next_url, last_good_url
    parsed_tup = urlparse( next_url )

    # Validate the parsed URL, if it is invalid return
    # with host = null
    # which will signal that another URL is needed

    if ( len( parsed_tup[1] ) == 0 ) :
        host=""
        return

    host = parsed_tup[1]
    doc = parsed_tup[2]
    params = parsed_tup[4]
```

Functions and groups of commands are denoted using space characters in Python. Java and C use a combination of braces `{ }`, commas, and semi-colons to denote groups of commands. In Python, the number of spaces before a command defines a group of commands. For example, the above `hostdoc_decoder()` function is defined using the `def` command and the function's commands are grouped by indenting each command with space characters. At first this spacing system may appear unusual, but its simplicity offers great benefits.

`hostdoc_decoder()` is a function that takes a URL and decodes it into the host name, document name, and parameters. For example, `hostdoc_decoder` would decode this URL:

```
http://examples.pushtotest.com/htmlresponder/
responder?file=file1.html
```

into these components:

- host: http://examples.pushtotest.com
- doc: /htmlresponder/responder
- params: file=file1.html

In the event that `urlparse()` finds an invalid URL, then `hostdoc_decoder()` sets the host variable to an empty string and returns:

```
httphandler = HTTPProtocol()
```

The `httphandler` variable now points to a new `HTTPProtocol` object. This object handles all communication with the HTTP/HTML Web-enabled application.

```
responselink = ResponseLinkConfig()
responselink.setParameter( 'beginsearch', f1 )
responselink.setParameter( 'endsearch', f2 )
```

These commands define a `ResponseLink` object to search for an <a href> hyperlink tag and will be used by the `HTTPProtocol` object when TOOL requests a Web page from the server. In this script, the `responselink` variable points to a new `ResponseLinkConfig` object. TOOL implements the `ResponseLink` object to handle search functions that parse through the responses received from the server.

The Wanderer agent wants to request a Web page, find a hyperlink to the next page, and continue reading pages. The agent accomplishes this by using a `ResponseLink` object to search for hyperlink tags in the HTML received from the server. Recall that previously the script defined variables f1 and f2 to contain the markers for a hyperlink. The setParameter method of the `ResponseLink` object tells the object to look for hyperlinks.

```
search = SimpleSearchLink()
search.init( responselink )
```

One last step is needed to create the search objects to actually find the hyperlinks in the server's response. While the `ResponseLink` object is used for searches of all protocols, the search variable now points to a `Simple-SearchLink` object that specifically searches HTML data.

To recap, the script creates several variables and search objects, and defines the `hostdoc_decoder()` function. The agent is ready to make HTTP requests to the server.

```
while 1:
    hostdoc_decoder( next_url )

    if host=="":

        # The host we picked isn't valid so
        # raise an exception

        raise Spider_Error( "Giving up!" )
```

The *while* causes the Wanderer to loop forever over a group of commands, which is because the value of 1 always evaluates to true. Other types of loops are possible. For example, instead the script could search through 1,000 Web pages using the *for i in range(1000):* syntax instead of a `while` command. But for the Wanderer life is eternal. (Of course, the handy Stop button will end the Wanderer's wanderings.)

```
    httphandler.setHost( host )
    if params == "":
        httphandler.setPath( doc )
    else:
        httphandler.setPath( doc + "?" + params )
```

We previously created an `HTTPProtocol` object that is referenced through the `httphandler` variable. Now, we need to tell it a few things before it can make a request to the server. First we must tell the object where to find the server. If the URL to the Web page includes parameters, we need to tell the `HTTPProtocol` object this using the setPath method.

```
    response = httphandler.connect()
```

The `connect` method tells the `HTTPProtocol` object to actually make the request to the server. The server's response is put into a new object that is referenced through the `response` variable. The `response` object provides us with many functions to learn about the server's response to the HTTP request.

```
found = search.handle( response )
foundcount = found.getParameterValue("simplesearch.foundcount")
```

The `response` object returns an object that holds the results of the search object, including a couple of the hyperlink tags found on the page.

```
foundlist = found.getParameterValues("simplesearch.founditems")
doc = foundlist.get( r.nextInt( foundcount ) )
```

The `found` object helps pick out the next Web page the Wanderer will request.

```
    next_url = "http://" + doc[ len(f1) : \
    ( len(doc) - len(f2) ) ]
```

The `doc` variable holds the HTML hyperlink tag. For example, the tag may look like this: <a href="http://examples.pushtotest.com/responder?file=file2 .html">. While the HTML tag is useful in displaying hyperlinks in a browser, the `hostdoc_decoder()` function only needs the URL. The next_url variable is set to the URL value by using a few of Python's String functions, including `len()`, which returns the length of a String and String slicing functions that use the bracket and colon characters.

```
if response.getTotalTime() > worsttime:
    worsttime = response.getTotalTime()
    worstname = last_good_url
```

The `response` object also provides values indicating the time it took to communicate with the server. The Wanderer script keeps track of the request that took the longest to complete.

```
worstcount = worstcount + 1
if worstcount > 10:
    print "===============Award time================="
    print "The award goes to: ", worstname
    print "which took ",worsttime," in milliseconds \
    to complete."
    print
    worstcount=0
```

For every 10 server responses, the Wanderer cruelly announces the results in an award ceremony. Not to worry though. All is forgiven until another 10 server responses are complete.

As we have just seen, the Wanderer test agent uses an `HTTPProtocol` object to get a Web page. It then finds hyperlinks on that page and follows a random hyperlink. The Wanderer also keeps track of the time it takes to receive each page. Just for fun the Wanderer pauses after every 10th-loaded Web page and gives an award to the Web page that took the longest to load. To the server, the Wanderer is a real user sitting in front of a browser making requests. When you run the Wanderer multiple times concurrently, the server responds to what it thinks are many concurrent users. The Wanderer is the first step at understanding how an HTTP/HTML Web-enabled application handles the load of many concurrent users to determine how that Web-enabled application will scale and perform under real production environments.

Understanding Cookies, Sessions, and Redirection

The Wanderer agent script is good at its role of creating load on the server by testing the relatively simple `GET` function of the Web-enabled application. However, dynamic HTTP/HTML Web-enabled applications provide personalized user experiences that also must be tested for functionality, scalability, and performance. Personalization usually requires session information so users only need to identify themselves once and subsequent Web pages display content that is personalized to them. These services require the user to sign in using an HTML form and then track the user with a session cookie. The Sign-in Agent shows how to work with sign-in forms and cookie-managed sessions.

Much writing and discussion has appeared in the popular media about cookies. Skeptics link cookies to water fluoridation, government cover-ups of alien landings, and collusion in the military–industrial–entertainment complex. In reality cookies are a value sent back to the browser in a server response to link a group of requests together. By design requests made using the HTTP protocol are independent of each other. Cookies provide a mechanism to link together HTTP requests. Figure 6–6 shows a series of requests using cookies to maintain a session.

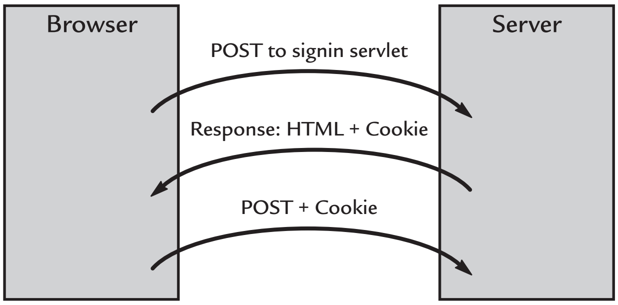


Figure 6–6 After the user signs in, subsequent requests and responses pass a cookie value to identify requests as part of an overall user session.

While early attempts at using cookies to monitor user activity on a Web site were made by a few public Internet sites, cookies today are mostly used to manage user sessions. The typical cookie may look like this:

```
Cookie: sessionid=38828x348v91
```

The cookie value will have meaning only to the server that created the cookie value. The value is often an index into a table maintained on the server to track the signed-in status of a group of users.

In this section we explore the Sign-in agent to see how Python and TOOL are used to work with HTML forms, sessions, and cookies. Below is the Sign-in agent in its entirety, followed by a detailed explanation of the Sign-in agent's components.

```
# Agent name: signin_agent.a
# Created on: May 15, 2002
# Author: fcohen@pushtotest.com

print "Agent running: signin_agent.a"
print "Description:"
print "  Web-enabled applications with HTML front-ends "
print "usually require session information"
print "  to know which features to offer to users. These"
print " services require the"
print "  user to sign-in using an HTML Form and then the"
print " service tracks the"
print "  user with a session cookie. This agent shows"
print " how to work with sign-in"
print "  and cookie-managed sessions."
print

# Import tells TestMaker where to find Tool objects
from com.pushtotest.tool.protocolhandler import \
ProtocolHandler, Header, Body, HTTPProtocol, \
HTTPBody, HTTPHeader

from com.pushtotest.tool.response import Response, \
ResponseLinkConfig, SimpleSearchLink

# Import useful Python and Java libraries
import sys
import java
from urlparse import urlparse

# hostdoc_decoder: Decodes a URL into the host name
# and document name

host = ""       # Holds the decoded host name from a URL
doc = ""        # and the document name from the URL
params = ""     # and the parameters of the call

def hostdoc_decoder( theurl ):

    global host, doc, params, http_ph

    # urlparse is a handy library function that returns
    # a tupple containing
    # the various parts of a URL, including host, document,
    # parameters, etc.
    parsed_tup = urlparse( theurl )

    # Validate the parsed URL, if it is invalid return
    # with host = null
    # which will signal that another URL is needed

    if ( len( parsed_tup[1] ) == 0 ) :
        host=""
        return

    host = parsed_tup[1]
    doc = parsed_tup[2]
    params = parsed_tup[4]

    http_ph.setHost( host )
    if params == "":
        http_ph.setPath( doc )
    else:
        http_ph.setPath( doc + "?" + params )

#    print "host=",host," doc=",doc," params=",params


# Main body of agent

print
"========================================================"
print "Step 1: Get the sign-in form"
print

# This URL returns an HTML page that contains a
# sign-in form

urlone = "http://examples.pushtotest.com/responder \
/htmlresponder?file=file2.html"

http_ph = ProtocolHandler.getProtocol("http")

hostdoc_decoder( urlone )

print "Sending request."

response = http_ph.connect()

# get an Iterator of all the keys
```

```
k = response.getParameterKeys();

for i in k:
    kv = k.next()
    print kv,":",response.getParameterValue( kv )
    print

print
"========================================================"
print "Step 2: Do a form submit to sign-in"
print

urltwo = "http://examples.pushtotest.com/responder \
/htmlresponder"
urlthree = "http://examples.pushtotest.com/responder \
/htmlresponder?file=file3.html"

hostdoc_decoder( urltwo )

http_ph.setType( HTTPProtocol.POST )

body = ProtocolHandler.getBody( "http" )

# send the sign-in values
body.addParameter( "login", "myname" )
body.addParameter( "passwd", "secret" )

# Next we use some of the responder servlets custom
# commands. Details of these commands are found
# on http://examples.pushtotest.com/responder/htmlresponder
# Tell the responder to return a redirect response
# code to urlthree
body.addParameter( "redirect", urlthree )

# Tell the responder servlet to set a cookie
body.addParameter( "cookie", "sessionid" )
body.addParameter( "cookievalue", "388281v90981" )

# Tell the responder to return the contents of file2.html
body.addParameter( "file", "file2.html" )

http_ph.setBody( body )

# We use a special form of connect() which tells the
# http protocol handler not to follow any redirect
# commands received from the host
```

```
print "Sending request."

response = http_ph.connect( 0 )

print
print "Here is what we got back from the host:"
print "response code =",response.getResponseCode()
print
print "Response parameters from the host:"

# get an Iterator of all the parameter keys

meb = response.getParameterKeys()

for i in meb:
    print i, "=", response.getParameterValue( i )

# Display the cookie name/value pairs
# received from the host

# Display the cookie name/value pairs received from the host
print
print "jCookie values:"
for i in http_ph.getCookies():
    print "bop=",i

print
print
print "============================================="
print "Step 3: Manually handle a redirect command "
print "from the server"
print

urlfour = response.getParameterValue( "Location" )

print "The host told us to redirect to:"
print urlfour
print

hostdoc_decoder( urlfour )

# In Step 2 we added a parameter that told the
# examples/responder servlet to send back a redirect
# command. Since we're using the same http_ph object
# that we used in Step 2 we need to tell the
# object to not send us the redirect this time.
# This situation is unlikely in a normal test but
# the following command is needed for this example agent.
body.removeParameter( "redirect" )

# Since we are reusing the http_ph object we used in Step 2
# this object already has the received cookie values from
# last host response and will play them back to the host

print "Sending request."

response = http_ph.connect()

print
print "Here is what we got back from the host:"
print "response code =",response.getResponseCode()
print
print "Response parameters from the host:"

# get an Iterator of all the parameter keys

meb = response.getParameterKeys()

for i in meb:
    print i, "=", response.getParameterValue( i )

print
print "Agent finished."
```

The Sign-in agent makes requests directly to the examples.pushtotest.com server. PushToTest, the principal maintainers of TestMaker, hosts this server. The Sign-in agent shows a typical group of commands that together provide a user sign-in function using HTTP/HTML protocols. The agent breaks these commands down into the following three steps:

1. Get the Sign-in form.
2. Do the `POST` command to sign in.
3. Receive a cookie and handle the redirect response from the server.

Let's investigate the individual parts that make up the Sign-in agent. The agent script begins as all good agents do by importing the TOOL, Python, and Java objects the agent will use, declaring a set of variables useful throughout the agent and declaring functions to be used by the agent.

```
def hostdoc_decoder( theurl ):

    global host, doc, params, http_ph

    # urlparse is a handy library function that
    # returns a tupple containing
    # the various parts of a URL, including
    # host, document, parameters, etc.

    parsed_tup = urlparse( theurl )

    # Validate the parsed URL, if it is invalid
    # return with host = null
    # which will signal that another URL is needed

    if ( len( parsed_tup[1] ) == 0 ) :
        host=""
        return

    host = parsed_tup[1]
    doc = parsed_tup[2]
    params = parsed_tup[4]

    http_ph.setHost( host )
    if params == "":
        http_ph.setPath( doc )
    else:
        http_ph.setPath( doc + "?" + params )
```

The Sign-in agent uses a modified version of the `hostdoc_decoder()` function that was first introduced in the Wanderer agent. This form of `hostdoc_decoder()` not only decodes the host, document, and parameters in a URL but also puts those values into a globally used `HTTPProtocol` object that is accessed through the http_ph variable.

```
urlone = "http://examples.pushtotest.com/responder \
/htmlresponder?file=file2.html"

http_ph = ProtocolHandler.getProtocol("http")

hostdoc_decoder( urlone )

response = http_ph.connect()
```

Step 1 of the Sign-in agent uses an `HTTPProtocol` object to get a Web page that contains the HTML sign-in form. In a browser this appears to the user as a place to enter their id and password and to click a sign-in button. The `response` object holds the response from the server, including a list of HTTP header parameters that will be of interest to us after we sign in.

```
k = response.getParameterKeys();

for i in k:
    kv = k.next()
    print kv,":",response.getParameterValue( kv )
    print
```

The `k` variable points to a list of the HTTP header parameters in the server response. Jython's `for` command makes it easy to go through the list and print out each header parameter value. The list contains the keys or titles of each parameter and the `getParameterValue()` function returns the value for a given key.

Next, the Sign-in agent makes a `POST` request to the server to sign in.

```
http_ph.setType( HTTPProtocol.POST )

body = ProtocolHandler.getBody( "http" )

# send the sign-in values
body.addParameter( "login", "myname" )
body.addParameter( "passwd", "secret" )
```

This part of the agent script reuses the `http_ph` `HTTPProtocol` object previously created. The `setType` method tells the `HTTPProtocol` object to issue a `POST` request to the server and the agent uses `addParameter` to add the sign-in login and passwd parameters to the body of the request.

Unlike most servers, the htmlresponder servlet on examples.pushtotest.com provides several features that are useful for illustrating what a real-world test agent might encounter. In particular, by sending a parameter named *redirect*, the server response will include a 302 Redirect command to the defined URL.

```
body.addParameter( "redirect", urlthree )
```

By sending the htmlresponder, a parameter named *cookie*, the server response will include a cookie parameter and value.

```
body.addParameter( "cookie", "sessionid" )
body.addParameter( "cookievalue", "388281v90981" )
```

And finally, by sending a parameter named *file*, the server will return the contents of a file found on the server.

```
body.addParameter( "file", "file2.html" )
```

The htmlresponder servlet includes additional commands not described here. For a full list of htmlresponder commands, see http://examples.pushtotest.com/responder/htmlresponder.

```
response = http_ph.connect( 0 )
```

In normal operation when the `HTTPProtocol` object in TOOL receives a 302 Redirect response, it would automatically request the redirect page. To illustrate a 302 Redirect response, the Sign-in agent uses a special form of the `connect()` method that will not automatically request the redirect page.

```
# Display the cookie name/value pairs received from the host
print
print "jCookie values:"
for i in http_ph.getCookies():
    print "bop=",i
print response.getParameterValue("Cookie")
```

The agent script then uses the `getCookies()` method to find a list of cookies being managed by the protocol handler. In the real world it is likely that an HTTP/HTML Web-enabled application would use more than one cookie at a time. TestMaker uses the jCookie library from Sonal Bansal to handle cookies and to provide you with several convenient methods to work with cookies. Details on jCookie are at http://docs.pushtotest.com. In general, you will likely not need to do anything with cookies as the `HTTPProtocol` object handles them for you.

The final step in the Sign-in agent uses the same `http_ph` object to request the redirect page. TOOL's `HTTPProtocol` object automatically sends the cookie values it received in the previous server response back to the server.

In this section we explored the Sign-in agent to see how Jython scripts and TOOL objects are used to work with HTML forms, sessions, and cookies. In the next section we will look at the Validator agent to see how server response data may be searched, parsed, and validated.

Validating Response Data

The previous two sections focused on building test agents that generate server load and conduct groups of commands in a cookie-based session. The Validator is an intelligent test agent that tests the server responses for valid data. The Validator shows four different techniques to search through server responses, including:

- Search and find response values using `ResponseLink` methods.
- Searching through response data using the scripting language commands.
- Parsing HTML forms using Tool commands.
- Finding XML data using JDOM commands.

Below is the Validator agent in its entirety, followed by a detailed explanation of the Validator's components.

```
# Agent name: checker_agent.a
# Created on: May 17, 2002
# Author: fcohen@pushtotest.com

print "Agent running: checker_agent.a"
print "Description:"
print "  This agent shows how to find useful
print "information in the response"
print "  received from host that returns HTTP/HTML
print "content. Three ways"
print "  of searching through the response are
print "presented, including:"
print "  1) Search & find using ResponseLink methods"
print "  2) Searching through response data using
print "the scripting language commands"
print "  3) Parsing HTML forms using Tool commands"
print "  4) Finding XML data using JDOM commands"

# Import tells TestMaker where to find Tool objects

from com.pushtotest.tool.protocolhandler import \
ProtocolHandler, Header, Body, HTTPProtocol, \
HTTPBody, HTTPHeader

from com.pushtotest.tool.response import Response, \
ResponseLinkConfig, SimpleSearchLink
```

```
# Import useful Python and Java libraries
import sys
import java
from urlparse import urlparse
from java.util import Random

# hostdoc_decoder: Decodes a URL into the host name and doc-
ument name

host = ""       # Holds the decoded host name from a URL
doc = ""        # and the document name from the URL
params = ""     # and the parameters of the call

def hostdoc_decoder( theurl ):

    global host, doc, params, http_ph

    # urlparse is a handy library function that
    # returns a tupple containing
    # the various parts of a URL, including host, \
    # document, parameters, etc.

    parsed_tup = urlparse( theurl )

    # Validate the parsed URL, if it is invalid return
    # with host = null
    # which will signal that another URL is needed

    if ( len( parsed_tup[1] ) == 0 ) :
        host=""
        return

    host = parsed_tup[1]
    doc = parsed_tup[2]
    params = parsed_tup[4]

    http_ph.setHost( host )
    if params == "":
        http_ph.setPath( doc )
    else:
        http_ph.setPath( doc + "?" + params )

#    print "host=",host," doc=",doc," params=",params


# Main body of agent
```

```
print
"======================================================"
print "Technique 1: Search & find using ResponseLink"
print

# This URL returns an HTML page that contains several
# <img> tags
urlone = "http://examples.pushtotest.com/responder \
/htmlresponder?file=file4.html"

http_ph = ProtocolHandler.getProtocol("http")

hostdoc_decoder( urlone )

# Establish search criteria

# Tool implements a simple system for searching
# and transforming an HTTP response. We begin by creating
# a ResponseLinkConfig object that defines the search
# parameters. In this case we're looking for <img> tags.

responselink = ResponseLinkConfig()
responselink.setParameter( "beginsearch", "<img" )
responselink.setParameter( "endsearch", '>' )

# Create a search object to handle the actual search.
# And we tell search to use the responselink.
# For example one uses the same
# ResponseLinkConf object to search through XML data when
# making SOAP calls.

search = SimpleSearchLink()
search.init( responselink )

print "Sending request."

# Connect to the host, get the document and return
# a response object
response = http_ph.connect()

# Find the next document URL in the body of the response
found = search.handle( response )

print "Here is what we learned from the SimpleSearchLink"
print
```

```
foundcount = found.getParameterValue("sim-
plesearch.foundcount")
print "Count of <img> tags:",foundcount
print

if foundcount > 0 :

    # This gets a list of the found items
    foundlist = found.getParameterValues \
    ("simplesearch.founditems");

    print "Here is a list of the <img> tags we found:"

    for i in foundlist:
        print i

    print
    print "We choose at random this <img> tag:"

    r = Random()    # A basic random number generator

    print foundlist.get( r.nextInt( foundcount ) )
    print


print "=============================================="
print "Technique 2) Searching through response data"
print "using the scripting language commands"
print

# We will use the <img> tags we already received
# in Technique 1 (above) to find the <img> tag that
# has the longest width value

if foundcount > 0 :

    # This gets a list of the found items
    foundlist = found.getParameterValues \
    ("simplesearch.founditems");

    # this keeps track of the widest <img> tag
    widest = 0
    widestkey = 0

    for i in foundlist:
        # There are 3 different conditions for finding
        # the width value in an <img> tag:
        # width=42   is the simplest, the value begins
        # after = and ends with a space
        # width="42" the value begins after the first
        # quote and ends with the second quote
        # width=42>  the value begins after the = and
        # ends with the >

        # first we get rid of those pesky " quote
        # characters
        widstr = i.replace( '"', '')

        # then we look for width=
        lexerr = "width="

        lexlen = len( lexerr )

        mark = widstr.find( lexerr )

        rest = widstr[ lexlen+mark : ]

        theend = rest.find( " " )
        if theend == -1:
            theend = rest.find( ">" )

        widthstr = rest[ 0 : theend ]

        if int( widthstr ) > widest:
            widest = int( widthstr )
            widestkey = i

    print "widest <img> tag = ", widestkey
    print

else:
    print "Didn't get any <img> tags in Technique 1,"
    print " so skipping Technique 2."
    print

print "============================================="
print "Technique 3) Parsing HTML forms using Tool commands"
print

# Technique 3 is going to use special Tool objects to
# handle HTML parsing
# and may possibly throw some Java exceptions
```

```
from com.pushtotest.tool.parser.html import \
HTMLParser
from java.net import URISyntaxException, \
MalformedURLException, URI
from java.io import IOException

urltwo="http://examples.pushtotest.com/responder \
/htmlresponder?file=file2.html"

try:
    uri = URI( urltwo )
    parser = HTMLParser( uri )
except URISyntaxException, ue:
    print "URLSyntaxException =",ue
    sys.exit()
except MalformedURLException, mfe:
    print "MalformedURLException =", mfe
    sys.exit()
except IOException, ioe:
    print "IOException =",ioe
    sys.exit()

forms = parser.parse();

print "This URL contains", len( forms ),"forms."
print

formnum = 0

for i in forms:
    formnum = formnum + 1

    print "Form",formnum,":"
    print
    print "This is an HTML form that uses a", \
    i.getMethod(),"command"
    print "to the host at:",i.getAction()
    print

    print "Here are all the <input> tags in the form:"
    print i
    print

    print "Here are the individual <input> tags:"
    for j in i.getElements():
        print "  name=",j.getName()," default"
        print " value=",j.getValueAsString()
    print

print
print

print "=============================================="
print "Technique 4) Finding XML data using JDOM commands"
print

# Technique 4 uses JDOM and Java StringReader objects to
# parse through XML data
from org.jdom.input import SAXBuilder
from java.io import StringReader

urlfour = "http://examples.pushtotest.com \
/responder/marketingfiles.xml"

hostdoc_decoder( urlfour )

print "Requesting XML document from host."
print

# Connect to the host, get the document and
# return a response object
response = http_ph.connect()

print "Here is the raw response document:"
print response
print

# Use the JDOM SAXBuilder object to create a new
# DOM object that
# will parse through the XML data we received
# from the host.

builder = SAXBuilder()
doc = builder.build( StringReader(response.toString()))

rootElement = doc.getRootElement()

print "The root element for this XML document is named:",
print rootElement.getName()
print
```

```
print "Here are the children of "
print " the",rootElement.getName(),"element:"

for i in rootElement.getChildren():
    print "  ",i.getName()

    # now let's show the children element
    # for each employee element
    for j in i.getChildren():
        print "    ",j.getName(),"=",j.getText()
    print

print
print "Agent finished."
```

The Validator agent makes requests directly to the examples.push-totest.com server. This agent shows four different techniques that validate, parse, and analyze the server response.

The first technique uses the TOOL `ResponseLink` object and methods to search through the server response. Much of this code has already been explained in the previous section on the Wanderer, so we focus on the other techniques in this agent.

The Validator agent looks for <img> tags in the HTML response from the server. These tags begin with <img and end with a closing > character.

```
responselink = ResponseLinkConfig()
responselink.setParameter( "beginsearch", "<img" )
responselink.setParameter( "endsearch", '>' )
```

The `ResponseLink` object defines the parameters of the search and a SimpleSearchLink object does the actual searching through the results.

```
search = SimpleSearchLink()
search.init( responselink )
```

The `HTTPProtocol` object returns the `response` object when it receives a response from the server.

```
response = http_ph.connect()
found = search.handle( response )
foundcount = found.getParameterValue("sim-
plesearch.foundcount")
```

From the `response` object, the agent accesses the found items from the `ResponseLink/SimpleSearchLink` duo. The `simplesearch.foundcount` value returns the number of items found in the server response.

```
r = Random()     # A basic random number generator
print foundlist.get( r.nextInt( foundcount ) )
```

Using the Java Random object in conjunction with the foundcount method, the agent can pick a found item at random. `ResponseLink` and SimpleSearchLink objects may be reused from response to response.

In the first technique the Validator made a request to the server and searched the response for <img> tags. This generated a list of <img> tags that appears as:

- <img src="images.pushtotest.com/logo.gif" width=401 height=58 border=0>
- <img src="images.pushtotest.com/empty.gif" width=10 height=1 border=0>
- <img src="images.pushtotest.com/man.gif" width=10 height=8 border=0>

Given this list of <img> tags, how would an agent find the widest image? The <img> tags are themselves simply strings of textual characters. Solving this problem requires the agent to parse through the results to find the width attributes in each <img> tag. Parsing HTML results data is a common problem when testing HTTP/HTML Web-enabled applications.

The Validator agent solves this parsing problem using the built-in Jython commands to work with strings.

```
foundlist = found.getParameterValues("simplesearch.founditems");
```

The agent begins by getting a list of the found items; these are the <img> tags in their entirety.

```
    widest = 0
    widestkey = 0
```

The agent establishes a variable named *widest* to hold the width of the widest <img> tag and a variable named *widestkey* to hold a pointer to the widest <img> tag as it scans through the list of tags. The agent script uses the powerful `for` command to loop through all the tags in the list. Each time through the loop, the script finds the width attribute in the tag and determines the numeric value of its width attribute.

HTML is a wonderfully flexible markup language. At times this flexibility can drive one to frustration. Consider that each of the following width attributes is valid in HTML notation:

```
width=42
width='42'
width="42"
width=42>
```

The Python language comes with a powerful `String` object to work with textual data. The `String` object implements dozens of methods for finding, replacing, and changing textual data. The agent script finds width attributes by first removing any quote and double quote characters using the `String` object's replace method. Then, the agent looks for the beginning of the width attribute using the find method. Finally, the agent uses a combination of `String` object methods to determine the starting and ending location of the width value. Using the `int()` operator, the agent converts the String of the width value into an Integer data type and compares it to the value in the variable named *widest*. At the end of the loop, the *widest* variable contains the width of the widest <img> tag and the *widestkey* variable holds a pointer to the widest <img> tag.

All this parsing of HTML content may appear to be quite error prone and it is! Each new advance that adds new forms of HTML tags makes it that much more difficult to parse through HTML to find the values an agent needs. And the same problem is true for any Web-enabled application that handles HTML content. Much of the fuel-driving development of XML technology comes from the frustration software developers feel when they work with HTML content. For more information about the software communities' response to HTTP/HTML Web-enabled applications, see the next chapter.

TOOL provides a simple HTMLParser object to find <form> tags in HTML pages. Technique 3 in the Validator agent uses the HTMLParser function to find <form> tags.

```
from com.pushtotest.tool.parser.html import HTMLParser
from java.net import URISyntaxException, \
MalformedURLException, URI
from java.io import IOException
```

The Import commands show where to find the HTMLParser object in TOOL. Additionally, we import several Java objects to handle error conditions that may result when we use HTMLParser. The script language provides a try/except mechanism to handle error conditions that may arise from using objects and methods. Code executes in a try block until an exception is encountered, at which time the program control switches to an except block.

```
try:
    uri = URI( urltwo )
    parser = HTMLParser( uri )
except URISyntaxException, ue:
    print "URLSyntaxException =",ue
    sys.exit()
except MalformedURLException, mfe:
    print "MalformedURLException =", mfe
    sys.exit()
except IOException, ioe:
    print "IOException =",ioe
    sys.exit()
```

For example, when the URI(urltwo) function is used, if urltwo contained an invalid URL, for example httx://blah.com, then the script execution would continue in the except MalformedURLException: block. As a result, the exception message is printed to the output window and the agent exits.

```
forms = parser.parse();
```

The parser.parse() method returns a list of <form> tags found in the parsed Web page. The Validator agent uses the for command to loop through each form. The contents of each form appear on the output window.

The last technique for validating server response data looks at parsing XML data in a server response. This should whet your appetite for SOAP-based Web-enabled applications that are covered in subsequent chapters.

```
from org.jdom.input import SAXBuilder
from java.io import StringReader
```

Technique 4 uses a Java object library called JDOM (http://www.jdom.org), which is a very Java-centric way of working with XML data, and the Java StringReader object to parse through XML data. JDOM is a Java-specific object-oriented interface to parsing XML documents. JDOM will appear in a future version of Java itself.

```
builder = SAXBuilder()
doc = builder.build( StringReader( response.toString() ) )
```

We begin by creating a new JDOM SAXBuilder object that will be referenced through the builder variable. SAXBuilder reads the XML document the agent receives in the server response into an internal Document Object Model (DOM) representation. In DOM format, the agent can use several XML search and manipulation methods to find the desired data.

The XML document looks like this:

```
<marketing>
  <employee>
    <id>582293</id>
    <firstname>Frank</firstname>
    <lastname>Cohen</lastname>
    <phone>408 374 7426</phone>
  </employee>
  <employee>
    <id>582343</id>
    <firstname>Betsy</firstname>
    <lastname>Hilbert</lastname>
    <phone>408 374 0000</phone>
  </employee>
</marketing>
```

The JDOM commands make it easy to find each of the elements, attributes, and values in an XML tree. For example,

```
rootElement = doc.getRootElement()
```

The `doc.getRootElement()` returns an object representing the <marketing> tag and all its children. Subsequent functions then find the children of the <marketing> tag, and so on. Given the choice of working with HTML or XML data, which would you choose?

Summary

This chapter covered the basics of HTTP and HTML technologies. We learned how TestMaker provides scripting language commands and a library of test objects to work with HTML forms, cookies, sessions, and complex response data. Finally this chapter covered several strategies for validating a

Web-enabled application's responses, including checking a response that is encoded in XML.

In a subtle way we charted the normal growth course of a software developer working on interoperating systems. The lure of HTTP/HTML Web-enabled applications is that you can develop powerful systems using this simple request-and-response protocol. It doesn't take long after the mastery of HTTP/HTML before the developer wonders why he or she cannot also move more complex requests and responses over the same network connection. In fact they can. And this realization ushered in the age of SOAP, WSDL, and UDDI-based Web-enabled applications that are the subject of the next chapter.

7 Tuning SOAP and XML Web Services

In the previous chapter, we began to see the natural progression that many software developers make when working on Web-enabled applications. Systems built on HTTP/HTML protocols and technology make use of HTTP's simple request-and-response protocol to move data from the desktop computer to a server and back again. The developer that masters building HTTP/HTML-based systems often wonders why the same network connection cannot move more complex requests and responses.

For example, when an HTTP/HTML-based application sends a credit card number from a browser to a server, the server needs special business logic to ensure the number contains the right number of digits. HTTP and HTML have nothing to define the type of data being sent and received. Unless the developer writes custom code, the server does not even know the entered value is a credit card number! Using HTTP, a credit card number and a phone number are the same—they are just a string of characters being sent to the server. What tells the server the difference between a credit card number and a phone number? This kind of developer thinking precipitated XML and Web Services.

Enterprises around the world had spent billions of dollars building systems that enabled their operations to be more efficient and streamlined. Now *integration* appeared to be the next great opportunity to increase the efficiency and productivity of businesses. In XML, not only does one send the data, but also a description that explains the type of the data. So a function already deployed within an enterprise could be identified through XML and made accessible to using existing HTTP/HTML Web Service infrastructures.

Exchanging data using XML-based protocols enables businesses to push their existing Web functionality outward to partners, vendors, suppliers, and customers. With XML, developers build systems by integrating functionality from several systems to deploy the next generation in information systems applications. XML is also recognized by developers as a means to integrate legacy and other non-Web-enabled applications into the Web Service model. As a result, software developers began to look for programming models that use XML in their existing Web infrastructures.

The early attempts at providing XML interoperability programming models concentrated on client/server-style architectures. The client software knew how to contact the server and made an RPC passing data encoded in an XML document. Thus was born XML-RPC. Details on XML-RPC are at http://www.xmlrpc.com/. A more generalized architecture was needed to reduce brittleness and improve maintainability between systems. SOAP and Web Service Description Language (WSDL) quickly followed. Details are at http://www.w3.org/TR/ws-desc-reqs/ and SOAP http://www.w3.org/TR/soap12-part0/.

XML-RPC and SOAP join a crowded field of software technologies that enable interoperability between services in an overall system. Consequently, a good understanding of the technologies and architectures is needed to deliver solutions that scale, perform, and function well under varied circumstances and environments. This chapter presents an entirely new set of software testing techniques and tools to find and solve XML-RPC and SOAP/WSDL-based Web Service problems, bugs, and scalability issues before users encounter them.

The Web Services Vision

Visit Web sites today and you are likely using more than one server software application package. This is especially true for dynamic Web sites offering personalized content. For example, when you buy a book on an Internet Web site, chances are the sign-in system will be hosted by one service, the order selection pages served by another service, and shipping and order tracking provided by a third system. If done well, the user encounters consistent branding and few or no seams between systems.

Behind the scenes, teams of engineers are doing their best to keep the backend systems interoperating. New hardware (routers, servers, storage) and revisions to server software (WebLogic, WebSphere, SAP, Oracle) appear with alarming frequency. Software engineers constantly write expensive and difficult-to-maintain custom software to keep the systems sharing data.

The invention of the SOAP and its younger brother, the WSDL, is our best hope yet to develop a common language for application-to-application communication. The benefits of being able to write applications that freely communicate with other applications, platforms, and hardware are many. Less time is spent on maintaining the code and the software developer gets to concentrate on solving bigger problems. The operations manager has a wider choice of platforms and hardware. Users benefit by having more advanced applications available. Overall, SOAP and WSDL deliver a language-independent, standards-based interoperability technology that runs on many different operating systems and application servers.

Software used to be something you bought at the local software store. It came in a plastic-wrapped, glossy oversized box containing a paper manual and a compact disc with a hundred megabytes of program code. Sometimes I feel nostalgic about these times since I had a lot of fun leading the teams that brought one packaged product after another to a software store near you—including Norton Utilities, Stacker, SoftWindows, and TuneUp. We wrote software to be delivered through a distributor or middleman, sold to users by a reseller, and installed on a single desktop computer. What could be simpler?

The software industry liked this for a while. Three or four word processors competed for your attention, same with spreadsheets and databases. These were products that engineers liked to write since they were self-contained software packages with very clearly defined boundaries. Unfortunately, for software developers, Microsoft's monopoly stranglehold on desktop software meant engineers rarely found investors willing to build companies around new word processors, spreadsheets, and databases—at least not for desktop computers.

Undaunted by the monopoly control of desktop application software, the next 10 years of technology innovations were primarily concerned with identifying the best location for the business logic and presentation libraries of a software application to run. Figure 7–1 shows how the first three generations of interoperable software architectures addressed this issue.

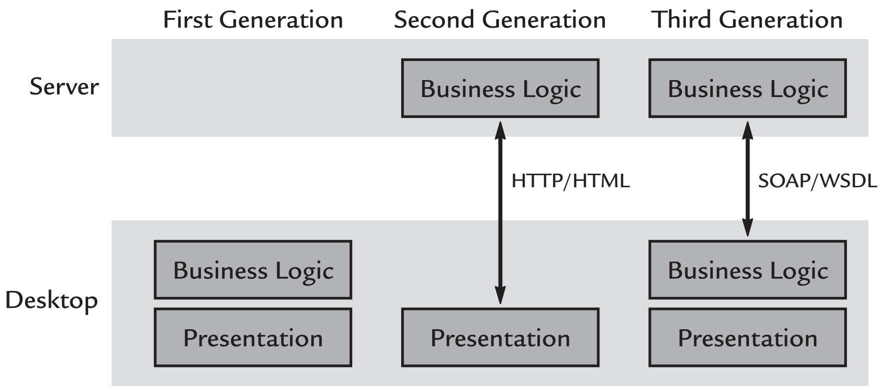


Figure 7–1 Software architectures have grown over time by changing the location of an application's business logic and presentation software.

Telecommunications industry deregulation of the late 1980s opened up the possibility of cheap modem connections between desktop computers. Over the next 10 years, publicly accessible networks became widely available and inexpensive. The engineers of the world rejoiced. They could write word processors, spreadsheets, and databases for servers and have the desktop computer act as a thin, dumb client for their code on the server.

By 2001, Microsoft reinvigorated the software development community by renovating its tools, server software, and operating system products with a renewed component model, pervasive support for XML, and a virtual machine technology that would be the core of their middleware technologies. Microsoft named the wide-ranging effort .NET. (Chapter 9 provides testing strategies and methodologies specifically for .NET.) Microsoft envisioned the component model to enable existing Microsoft word processor, spreadsheet, and database software on the desktop computers to interoperate with other word processor, spreadsheet, and database software on other desktops using XML Web Services, sometimes without the need for a server. This confused and emboldened engineers to try something new, as shown in Figure 7–2. Engineers could build and deploy solutions where the computing power is on the desktop, in a smart appliance, and in the server. Software developers, QA analysts, and IT managers need techniques and strategies to deal with these changing architectures.

Microsoft went further in its vision for .NET when it adopted support for UDDI. UDDI services enable software components to discover business

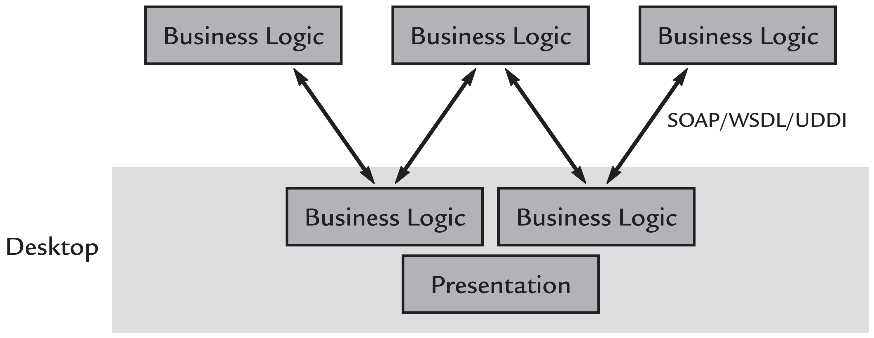


Figure 7–2 With the world supporting HTTP/HTML Web infrastructure, SOAP becomes the common language for a world of interoperating Web services running on a desktop and communicating with several content sources.

logic in other software components. The UDDI specification allows software modules to be categorized by function. Combining SOAP, WSDL, and UDDI enables software components to find, bind to, and remotely call the functions a component needs to complete its work. At the time of writing, few if any lexicographers and taxonomy experts were working on UDDI systems, so the basic function of categorizing and finding functions is not yet realized.

Competing with Microsoft, the providers of Java platforms, application servers, and tools were ready to incorporate XML too. SOAP and WSDL rapidly gained acceptance. Each company benefited from SOAP/WSDL in their own way. IBM used SOAP to XML enable its customer's legacy software, including COBOL and FORTRAN applications. BEA was the first to use the Java Web Service (JWS) standard to make it easy for Java developers to expose existing classes and methods as SOAP interfaces. SOAP gained broad platform-neutral industry support rapidly. Details on Microsoft's latest moves in Web services may be found at http://msdn.microsoft.com/Webservices/.

You need a good understanding of XML-RPC and SOAP/WSDL-enabled systems to understand how to test Web service-based systems. Let us begin with the XML-RPC protocol.

XML-RPC for Web Interoperability

XML-RPC is an open standards specification that describes how to write software that will make remote procedure calls using HTTP as the transport

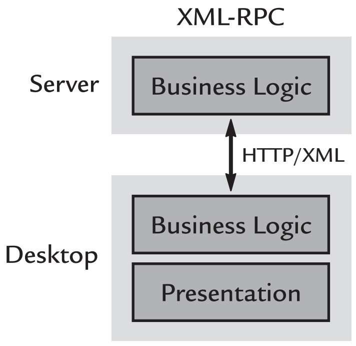


Figure 7–3 Using the sample infrastructure as an HTTP/HTML request, XML-RPC sends XML-encoded data to the service.

and XML as the encoding. A Web-enabled application decodes the XML and makes a call to a method that exists on the server. XML-RPC (http://www.xmlrpc.com/spec) was designed to be as simple as possible while still enabling complex data structures to be transmitted, processed, and returned. The specification's simplicity makes it possible for XML-RPC libraries to be implemented on a huge array of operating systems and programming languages, including Perl, Python, Java, Frontier, C/C++, Lisp, PHP, Microsoft .NET, Rebol, Real Basic, Tcl, Delphi, WebObjects, and Zope. Lest anyone forget, the first implementation of XML-RPC was on Userland Software (http://www.userland.com/) Frontier scripting-based application framework, which we will see later in this chapter as important to XML-RPC's evolution into SOAP.

An XML-RPC request to a Web service is formed much like an HTTP/HTML `POST` command is crafted. Instead of using HTML form elements, the body of the request contains the XML-encoded data to be sent, as shown in Figure 7–3.

A simple XML-RPC service is set up on the public Internet at examples.pushtotest.com. This service takes two parameters encoded in an XML document and echoes the parameter values back in an XML response document. Using a computer network monitoring utility illuminates the data that is transmitted and received in an XML-RPC call. (Later in this section I provide URLs to popular network monitors.) Here is the request going to the service:

```
POST./.HTTP/1.1
Content-Length:.240
```

```
Content-Type:.text/xml
User-Agent:.Java1.4.0
Host: mayo.pushtotest.com:91
Accept: text/html,.image/gif,.image/jpeg,.*;.q=.2,.*/*;q=.2
Connection:.keep-alive

<?xml.version="1.0".encoding="ISO-8859-1"?>
  <methodCall>
    <methodName>echo</methodName>
    <params>
      <param>
        <value>Tell.me.what.I.would.like.to.hear.</value>
      </param>
    <param>
      <value>Whisper.sweet.nothings.in.my.ear.</value>
    </param>
  </params>
</methodCall>
```

To make the request data more legible, I changed the spacing of the request data. In the real request, only carriage return characters separate all the XML and HTTP header data. Otherwise, this is the entire request. It should look similar to an HTTP/HTML request since XML-RPC is operating over the same Web infrastructure of transport protocols and routed networks. XML-RPC requires business logic to exist in the software that runs on both ends of the connection. The business logic can be implemented in a browser using JavaScript, Applets, or ActiveX objects or in a desktop application using one of the XML-RPC libraries.

XML-RPC responses also look similar to an HTTP/HTML response but contain XML data, as shown below.

```
HTTP/1.1.200.OK
Server:.Apache.XML-RPC.1.0
Connection:.Keep-Alive
Content-Type:.text/xml
Content-Length:.247

<?xml.version="1.0".encoding="ISO-8859-1"?>
<methodResponse>
    <params>
      <param>
        <value>
          <array>
            <data>
```

```
              <value>Tell.me.what.I.would.like.to. \
              hear.</value>
              <value>Whisper.sweet.nothings.in. \
              my.ear.</value>
            </data>
          </array>
        </value>
      </param>
    </params>
</methodResponse>
```

This response uses standard ISO-8859-1 encoding that handles Unicode characters among other things. In the response is an array containing the two parameters we transmitted in the request. Since this is XML, the response data can contain simple as well as complex data types. A very nice tutorial on ISO-8859-1 encoding is found at http://www.cs.tut.fi/~jkorpela/chars.html.

Viewing the HTTP-encoded protocol data as it moves across the network connection between the browser and the server may help you build and debug intelligent test agents. Windows NT, 2000, and XP users may view network traffic with the Microsoft Network Monitor utility. This utility captures the packets of data and displays them to show the `GET` and `POST` commands and their associated data values. Linux users may view network traffic with the popular tcpdump (http://www.tcpdump.org), ethereal (http://www.ethereal.com/), and NetWatch (http://www.slctech.org/~mackay/netwatch.html) utilities.

Writing a TestMaker test agent script to request a function over an XML-RPCprotocol is straightforward. Here is the script in its entirety, followed by an explanation of how it works. All of the code presented in this book is also available for download at http://www.pushtotest.com/ptt/thebook.html.

```
# Agent name: XMLRPC_Connect.a
# Author: fcohen@pushtotest.com

thehost = "examples.pushtotest.com"
theport = 91
request1 = "Tell me what I would like to hear."
request2 = "Whisper sweet nothings in my ear."

print "Agent running: XMLRPC_Connect"
print "Description: This agent will connect to a demonstra-
tion XML-RPC host"
print "on the PushToTest server. The host will echo back
whatever the agent sends."
```

```
print
print "Sending these values to the XML-RPC host on: ", the-
host, " port: ", theport
print "request1 = ", request1
print "request2 = ", request2
print

# Import needed TestMaker, TOOL and Jython libraries
from org.apache.xmlrpc import XmlRpcClient
from com.pushtotest.tool.protocolhandler import ProtocolHan-
dler, Header, Body, XMLRPCProtocol, XMLRPCBody, XMLRPCHeader
from com.pushtotest.tool.response import Response

# First we get an XMLRPC protocol handler object
protocol = ProtocolHandler.getProtocol("xmlrpc")

# Then we put together the request document
body = XMLRPCBody()

body.setTarget( thehost )
body.setTransport("http")
body.setMethod("echo")
body.addParameter( request1 )
body.addParameter( request2 )
protocol.setBody( body )

protocol.setHost( thehost )
protocol.setPath( "" )
protocol.setPort( theport )

print "Connecting..."

response = protocol.connect()

print "Here is the response from the host:"
print response

print
print "That request took ", response.getTotalTime(), " mil-
liseconds to complete."
print
print "Agent ended."
```

If you followed the example agents in Chapter 5, then testing XML-RPC services should look just like testing HTTP/HTML applications. Next I describe this test agent script in detail.

```
from org.apache.xmlrpc import XmlRpcClient
from com.pushtotest.tool.protocolhandler import ProtocolHan-
dler, Header, Body, XMLRPCProtocol, XMLRPCBody, \
XMLRPCHeader
from com.pushtotest.tool.response import Response
```

The test agent begins by importing the needed TestMaker objects to communicate with a Web service using XML-RPC protocols.

```
protocol = ProtocolHandler.getProtocol("xmlrpc")
```

The test agent uses an XML-RPC protocol handler and body object. These objects will handle the actual conversation with the Web service using HTTP protocols and transmitting and receiving encoded XML data.

```
body.setTarget( thehost )
body.setTransport("http")
body.setMethod("echo")
body.addParameter( request1 )
body.addParameter( request2 )
```

The body object determines the URL to the Web service and the name of the method to be called. Adding the parameters to the body enables the TestMaker protocol handler to encode the request1 and request2 strings into an XML request document that are sent to the Web service for processing.

```
response = protocol.connect()
```

The `response` object contains the XML response from the Web service. Printing the `response` object to the output window shows the XML data. TestMaker includes XML handling libraries to make it easy to parse through the XML response document to validate the Web service response. Details on validating the response come later in this chapter. For an example of how to construct the Web service on the server, see http://www.pushtotest.com/ptt/kits/index.html.

Where XML-RPC Falls Short

The simplicity in XML-RPC belies a bigger architectural problem: interoperability in XML-RPC systems is extremely brittle. Consider the following issues:

- XML-RPC defines no clean mechanism to pass XML documents themselves in an XML-RPC request or response. Since the request and response is itself an XML document, what happens when you need to send XML-encoded data to a service for processing?
- There is no way to programmatically change the format of a request or response document type. Parameters and methods normally change when software is maintained. Any change to the client or the service requires a change in the method on the client and server side to recognize the change. This makes XML-RPC brittle.
- How will XML-RPC be extended to support security, encryption, data compression, and a host of other special needs for vertical industries? XML-RPC covers only the basic HTTP/XML protocol today.
- XML-RPC does not support namespaces. Every field name needs to be totally unique from the others or else the XML-RPC client or server will get confused. This is not usually a problem for a request that has a few parameters, but imagine each of 1,000 response elements needing unique field names.
- XML-RPC operates exclusively over HTTP. Several innovative Web service-based system designs have passed RPC calls through message queues, databases queries, and even email.

The ever-vigilant software industry has topped XML-RPC with a new invention: SOAP.

Universal Interoperability with SOAP

Userland Software was an early and vocal proponent of XML-RPC and led its creation as a standard. XML-RPC is wonderfully simple to learn and use, and inexpensively operates over existing Web infrastructures. However, XML-RPC becomes brittle when Web-enabled applications change and need maintenance. Userland was in the unique position to convince Microsoft, IBM, and later Sun to build a new interoperability standard that began with XML-RPC and solved the expandability and brittleness problems. The resulting work is the SOAP.

The SOAP protocol uses a multistep process to complete a transaction, as shown in Figure 7–4. In a SOAP transaction the business rules of a software application make a request to a SOAP method on a remote server. The request contains parameters that are converted from the native language (for example, Java, C, C++, and Visual Basic) into a string of characters by a set of serializers. The serialized parameters become part of an XML document that conforms to the SOAP Envelope specification. A SOAP Envelope contains a header and body. The body holds the XML-encoded serialized parameters. The SOAP Envelope is transmitted to the Web service using HTTP or SMTP protocols. Additional transports are available, such as RMI and IIOP, but those are rarely used today.

Soon after SOAP's invention, Microsoft and IBM became the leading proponents of WSDL as the standard metalanguage to describe SOAP-based Web services. WSDL does an adequate job of describing the methods,

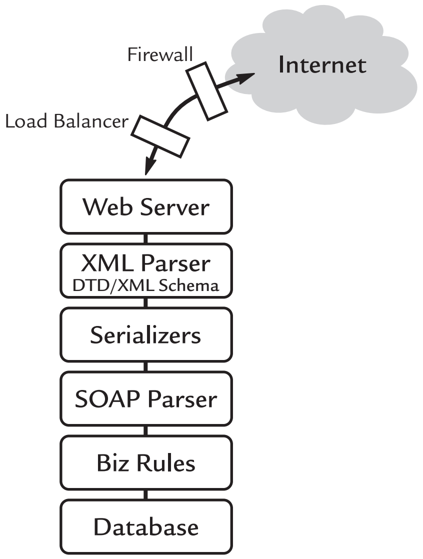


Figure 7–4 SOAP requests begin in the Business Rules of a Web-enabled application and use a combination of technologies to encode requests into XML data to be transmitted through a Web infrastructure.

parameters, and data types used to make a SOAP request and receive a response. WSDL is machine parsable and enables development tools and application servers to generate program source code.

SOAP provides better extensibility and reduces brittleness over XML-RPC by introducing the many extra layers of the SOAP stack described in Figure 7–2. However, with greater flexibility comes a greater possibility of incompatibility and scalability problems.

Web Service Scalability Techniques

SOAP and WSDL-based Web services use a multistep process to complete a transaction. Many techniques and system architectures attempt to improve Web service scalability and performance. Understanding these techniques is important to validate the results in a test.

The Web service request often begins with business logic of your application learning the method and parameter to call from a WSDL document. As an example, here is part of the WSDL for a publicly available Web service that returns the current weather for a U.S. postal zip code.

```
<message name = "getTempRequest">
 <part name = "zipcode" type = "xsd:string"/>
</message>
<message name = "getTempResponse">
 <part name = "return" type = "xsd:float"/>
</message>
```

The weather service requires you to call the getTempRequest method by passing in a zipcode value as a string and receiving the temperature as a floating-point value in the response.

Since the WSDL rarely changes, many developers embed the WSDL definition into their code to avoid the overhead of getting the WSDL every time. While this will improve performance, it also works against solving brittleness and becomes a maintenance headache when the WSDL eventually changes. The better way to avoid maintenance problems is to cache the WSDL in the centralized database and then periodically check the timestamp/version number of the WSDL to see if a newer one is available.

Another way for software developers to try to improve performance is to turn XML validation off. For systems that do no use validation, test suites should use light validation of the response results. For example, this WSDL defines the schema for the response:

```
<element name="zipcode" type="int"/>
<element name="temperature" type="float"/>
<element name="remarks" type="string"/>
```

The result of a call to this service looks like this:

```
<zipcode>95008</zipcode>
<temperature>65 F</temperature>
<remarks>Storm warning</remarks>
```

This response should throw an exception because the temperature value is not a float type. It is actually a string. Validating the response in an intelligent test agent will normally be much faster than depending on the DTD or XML Schema code to validate the SOAP response.

Parameter types in SOAP present a possible scalability problem too. SOAP defines simple data types: String, Int, Float, and NegativeInteger. The simple data type such as a String appears in WSDL using XML Schema like this:

```
<message name = "getTempRequest">
 <part name = "zipcode" type = "xsd:string"/>
</message>
```

As we will see later in this chapter, the SOAP request and response may include non-trivial new data types. For example, imagine the temperature Web service also retrieved maps. The schema for the call may look like this:

```
<message name = "getTempRequest">
 <part name = "zipcode" type = "xsd:string"/>
</message>
<message name = "getTempResponse">
 <part name = "return" type = "xsd:float"/>
 <part name = "map" type = "xsd:http://www.pushtotest.com/
wsdl/mapformat"/>
</message>
```

While reading the response, a validating XML parser will contact the pushtotest.com host to get the XML Schema definition for the mapformat. The overhead of this request can cause scalability problems when the validating parser does not cache the schema definitions.

A general performance rule is to stay with the simple SOAP data types unless there is a compelling need to use another data type. Each new data type introduces a serializer to convert from the XML value into the local programming language (Java, C, C++, and Visual Basic) value and back again. The serializer may cause performance problems or just be buggy. For example, the Apache and Microsoft SOAP implementations both include a BigDecimal data type. However, prior to Java 1.3 and .Net 1.2 the two are not compatible.

While SOAP was designed to work within existing Web application environments, the protocol may introduce firewall and routing problems. Unlike the normal Web server using HTTP, SOAP messages using HTTP as a transport are the equivalent of HTTP Form Submits. The calls move much more data than the average HTTP `GET` or `POST`. This is bound to impact network performance. Special testing of the firewall and routing equipment should be undertaken. For example, it is prudent to check your firewall's security policy to make certain it does not monitor SOAP requests as Web traffic. If it does you may find the firewall shunting away traffic that looks like a Denial of Service (DOS) attack.

The early Web services are very straightforward: you make a SOAP call and get a response. More advanced SOAP applications make a series of get and response calls until a transaction is finished. Transactional SOAP calls need to identify sessions and cache the state of the sessions. Caching mechanisms for SOAP transactions are potential problem spots for scalability.

Web Service Interoperability Problems

Stepping onto the new Web services island, one might think your problems are behind you. Then the reality of Web services sets in. Dozens of platform providers, independent software vendors, and utility software developers have implemented SOAP and WSDL in their products. Many times developers have had to interpret the meaning in parts of the specifications. Interpretation allows interoperability problems to seep into SOAP-based Web services.

Web service interoperability goals are to provide seamless and automatic connections from one software application to another. SOAP, WSDL, and UDDI protocols define a self-describing way to discover and call a method in a software application, regardless of location or platform. Data is marshaled into XML request and response documents and moved between software packages using HTTP or message-based protocols. Interoperability problems, such as platform-specific differences in BigDecimal, creep-in at the discovery, definition, and request/response mechanisms.

Discovery

In the dreamy world of Web Service Utopia, every software application is coded with a self-discovery and self-categorization method. Software that lacks a needed function looks into a UDDI-based registry of services and automatically makes a contract with a found Web service to handle the task. WSDL and SOAP enable communication once a Web service function is found. The problem then is to categorize what the function does so it may be found. UDDI defines TModels that are taxonomies to describe the location, path, and character of the function.

UDDI enables businesses to host online registries of available Web services. On the public Internet, Microsoft, HP, and IBM offer UDDI registries to businesses. A business uses the UDDI TModel system to categorize the hosted Web service. And therein lies the problem: UDDI allows multiple taxonomy and expects self-policing for mistaken entries in the registry. For example, suppose a Web service that prints and sends invoices lists itself in a UDDI registry using an SIC code but does not list geographic information. Using such a Web service from the other side of the planet would work; however, it may be easier to lick the stamp yourself.

In time UDDI will be well used and understood by the traditional taxonomy providers, including LCSH (Library of Congress Subject Heading), FAST (Faceted LCSH), DDC (Dewey Decimal Classification), and LCC (Library of Congress Classification). Until the taxonomy experts add their practical knowledge of developing and maintaining public directory structures in UDDI, you should plan for interoperability problems.

On the other hand, private UDDI directories are already viable. Enterprises have spent billions of dollars renovating their supply-side systems. With these renovations comes standardization of product definitions and business processes. These processes can be easily moved into a UDDI registry and accessed from UDDI-enabled Web services.

Definition

Web services uses WSDL to define how to make a request to a SOAP-based method. WSDL assumes cooperation from companies that define custom data types. Such cooperation is put to the test by collaborative organizations that are establishing interoperability test suites. SOAPBuilders is a loose affiliation of Web service vendors whose goal is to proof interoperability. SOAP-Builders also publishes an interoperability test suite for checking SOAP implementations that is available at http://www.xmethods.com/ilab/ and http://www.whitemesa.com/interop.htm. The test suites emerging today begin with the WSDL definition of a SOAP interface. They test the contents of the request and response documents for valid data.

This has put renewed energy behind WSDL efforts. New technologies, such as Cape Clear CapeStudio and BEA WebLogic Workshop, automatically develop WSDL documents for SOAP-based Web services. Tools like these eliminate poorly constructed WSDL that appears when developers hand code WSDL documents. Plus, Java itself is getting much better at handling WSDL in the Java Web Services Developer Package. Details are at http://java.sun.com/webservices/.

```
<?xml version="1.0" ?>
<definitions name="PushToTestService" targetNamespace
="http://www.pushtotest.com/pushtotestservice.wsdl">
<message name="testRequest">
  <part name="userName" type="xsd:string" />
  <part name="authenticationToken" type="xsd:string" />
  <part name="goodUnitl" type="xsd:Date" />
  </message>
. . .
```

WSDL documents have been known to cause interoperability problems. For example, consider the above snippet of WSDL for a software test Web service. The WSDL defines how to send a testRequest command; however, the <definitions> element fails to define the name space. The correct <definitions> element should look like this:

```
<definitions xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:s0="http://tempuri.org/" targetNamespace="http://tem-
puri.org/" xmlns="http://schemas.xmlsoap.org/wsdl/">
```

The developer likely thought the Web service would default to the standard W3 SOAP name space. While this may work for primitive data types like String, there are known interoperability problems with Date data types that appear later in this chapter. Without specifying the namespace, the Web service will likely fail to handle the data type correctly.

Request/Response

SOAP defines a standard way for software applications to call each other's methods and to pass data. SOAP requests are XML documents containing a description of the namespace, method called, and data. XML tries to be fairly flexible to allow developers to write XML elements and definitions. The flexibility can be a problem for SOAP interoperability.

For example, a typical SOAP response document might look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:xsd="http://www.w3.org/2001/
XMLSchema" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
<SOAP-ENV:Body>
<ns1:echoStringResponse xmlns:ns1="http://soapinterop.org/">
<result xsi:type="xsd:string">Hello!</result>
</ns1:echoStringResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This response document sends back a String containing the text "Hello!" The <result> element also includes the xsi:type="xsd:string" parameter that gets deserialized into a native language object (for example, a Java `String` object). Many SOAP tools add explicit typing information into the request and response document. On the other hand, some SOAP libraries return responses with no type information.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:xsd="http://www.w3.org/2001/
XMLSchema" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
<SOAP-ENV:Body>
<ns1:echoStringResponse xmlns:ns1="http://soapinterop.org/">
<result>Hello, I'm a string!</result>
</ns1:echoStringResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The <result> element in the above response document includes no type information. In this case, the SOAP library that deserializes the <result> value must look into the WSDL description of the service to find the description of the return type. If the WSDL does not define a response type, then it should throw an exception. However, in my experience, more times than not the SOAP library will default to a `String` object and that may be incorrect.

Data types are where the rubber meets the road in Web services. SOAP uses serializer and deserializer objects to translate from the native language of a software application to the SOAP protocols that move the request over the wire. It is here where native languages introduce dependencies on the data. For example, the way Java defines date objects is different than Microsoft .NET C++ date objects. This has the unfortunate effect of allowing SOAP data types with the same name to have different implementations; thus, interoperability problems lie ahead.

The most common data types to fail interoperability tests are Floating Point numbers and Dates.

Floating-Point and Decimal Data Types

Floating-point numbers in SOAP are represented as strings of decimal digits. The SOAP definition for floating point numbers also enables scientific notation, for example 7.53E+10, to handle exponential numbers using notation in use by engineers for decades. In general this works as expected; however, when pushed, floating-point numbers have problems.

For example, the original IBM SOAP4J implementation (now the Apache SOAP and Apache AXIS libraries) used the Java toString method and constructor to convert floating-point values found in SOAP documents into Java objects. When it came to serializing the floating point number "infinity," Java outputs the string as "Infinity." On the other hand, XML Schema serializes infinity as "INF." This caused SOAP4J to have interoperability problems with other SOAP toolkits.

Just as the Internet was born from the cooperation of network administrators, today we see SOAP implementers cooperating to solve interoperability problems. Apache SOAP, the successor to SOAP4J, was changed to accept "INF" as a valid way to serialize infinity.

Decimal data types suffer from language dependencies when pushed too. Decimal data types may represent large numbers up to 40 digits of precision. Relying on all 40 digits in a SOAP request or response is problematic unless both server and client are implemented on the same language. This is also true for fractional seconds with dates and trailing zeros on decimals.

BigDecimal is a good example of interoperability pitfalls introduced by language dependencies. Decimal numbers are a necessary part of financial calculations in banking applications where huge numbers are required. The XML Schema specification for decimal data types allows an arbitrary precision. Decimal data types could represent 1,000 digits of precision—that is, a decimal number represented as a string of 1,000 one-digit numbers. Apache SOAP is based on the Java implementation of the BigDecimal data type. Java's BigDecimal has an upper limit to precision of a number depending on the underlying operating system (Solaris, Windows, etc.)

XML Schema responded to these kinds of interoperability problems by defining a minimally conforming implementation specification. In the case of decimal data types, XML Schema requires at least 18 digits of precision. Apache and Java meet the requirement. But that does not mean a SOAP-based Web service will receive the minimum precision.

Microsoft .NET implementations handle BigDecimal data types up to 29 digits of precision. So, what happens to the extra digits of precision when an Apache SOAP request with a BigDecimal data type receives a .NET response? Unfortunately, it rests with the local SOAP serializer and deserializer implementation to know. And there is the rub: The SOAP transaction is valid but the data is wrong. Developers need to be vigilant enough to code data tests and protections into their software applications to consider invalid data from a SOAP exchange.

Given enough time and energy, these kinds of interoperability problems are solved by the Web Service platform providers. For example, Sun updated Java to provide BigDecimal interoperability with the .NET platform. From the very nature of what we do at work, we software developers, QA technicians, and IT managers are on the front-line to encounter these kinds of problems first. Being vigilant counts.

Date Data Types

Date data types suffer from interoperability problems to a greater extent than floating point decimal data types. The XML Schema defines the dateTime data type to contain centuries, years, months, days, hours, minutes, and seconds. But what about milliseconds, microseconds, and even smaller measurements of time? We live in a world where 2 GHz Intel CPUs sell for less than $150 and users require Web services to perform in 2–3 seconds at most. In just about every measurement of Web Service performance, milliseconds are going to count.

XML Schema specifies that any number of digits after seconds may be coded into a dateTime data type, but there is no minimum number of digits that an application must support. Apache SOAP uses the Java Date class (java.util.Date) to serialize and deserialize dateTime data types. Java Date supports precision up to the nearest millisecond. .NET's Date data type uses subsecond values up to four digits of precision—so a nanosecond may be represented in a .NET Date data type.

On the Horizon

Today, Web services are provided by the core UDDI, WSDL, and SOAP protocols. On the immediate horizon are a second layer of protocols that define workflow automation, Web service management services, and vertical market protocols. Web services greatly help developers build highly integrated solutions. So it should be no surprise to see interoperability problems arrive when workflow automation Web services are mixed with vertical market Web services.

If Web service toolkits are continually improved to solve interoperability problems, then customers, users, and businesses will solve system integration problems more efficiently than when using the existing standards (CORBA, DCOM, and RMI). The more serious Web service toolkit vendors have been diligent at solving interoperability problems. If interoperability problems linger or get worse, then we are in for slower adoption and much larger professional services costs to implement integrated systems. In the meantime, Web services are off to a great start.

Using TestMaker to Understand WSDL

The WSDL is a specification that describes the parameters, methods, data types, and accessors to a SOAP-based Web service. While it is possible to write SOAP-based Web services without WSDL, the benefits of having a metalanguage to describe the SOAP interface to a method are worth the extra effort.

Many times a test strategy is facilitated by having a utility retrieve and understand the WSDL document that describes a SOAP-based Web service. Now, let us see how TestMaker works with WSDL documents. Following is an agent that reads and parses a WSDL definition for a SOAP-based Web service running on the public Internet at examples.pushtotest.com. The agent is shown in its entirety and then is followed by an explanation of how it works.

```
# Agent name: explore_wsdl.a
# Author: fcohen@pushtotest.com

# --------------------------------------------------
# Set parameters and variables for this agent

# Location of the WSDL for the responder Web Service
target_wsdl = \
"http://examples.pushtotest.com:92/axis/servlet/ \
AxisServlet/responder_rpc?wsdl"


# Import tells TestMaker where to find Tool objects
from com.pushtotest.tool.parser.wsdl import WSDLParser
from javax.wsdl import WSDLException


# Import useful Python and Java libraries
import sys
import java
from urlparse import urlparse
from java.util import Random


# Main body of agent

try:
    parser = WSDLParser( target_wsdl )
definition = parser.getDefinition()
except WSDLException, ex:
    print "Something went wrong trying at:"
    print target_wsdl
    print
    print "The complete exception is:"
    print ex
    sys.exit(1)

print
print "Web Service Description"
print
print "At this location:"
print target_wsdl
```

```
services = definition.getServices()

for keys in services.keySet().iterator():

    current = services.get( keys )
    print "there is a Web Service named", \
    curent.getQName().getLocalPart(), "offering \
    these services:"
    print

    ports = current.getPorts()
    for pkeys in ports.keySet().iterator():
        pcurrent = ports.get( pkeys )
        bind = pcurrent.getBinding()
        print "Service:", pcurrent.getName()
        print "Binding:", bind.getQName()
        for extenelem in \
        pcurrent.getExtensibilityElements():
            print "Target URI:", extenelem.getLocationURI()

        print
        print "The",pcurrent.getName(),"service offers \
        these methods:"
        print

        for ops in bind.getBindingOperations():
            if ops.getName()!="class$":
                print ops.getName(), "which needs these \
                parameters:"
                print

                mcur = \
                ops.getOperation().getInput().getMessage()
                if mcur != None:

                    parts = mcur.getParts()
                    for partkeys in \
                    parts.keySet().iterator():
                        part = parts.get( partkeys )
                        print "  ",part.getName(), "that \
                        is of type", part.getTypeName()
                        print

    print

print
print "Agent finished."
```

Running this agent produces a nicely formatted report of the information offered in the WSDL document for the SOAP-based Web Service. Following is a portion of the agent's output:

```
Web Service Description
At this location:
http://examples.pushtotest.com:92/axis/servlet/AxisServlet/
responder_rpc?wsdl
there is a Web Service named responder_rpcService offering
these services:
Service: responder_rpc
Binding: http://examples.pushtotest.com:92/axis/servlet/
AxisServlet/responder_rpc:responder_rpcSoapBinding
Target URI: http://examples.pushtotest.com:92/axis/servlet/
AxisServlet/responder_rpc
The responder_rpc service offers these methods:
Callback which needs these parameters:
   host that is of type http://www.w3.org/2001/
XMLSchema:string
   times that is of type http://www.w3.org/2001/
XMLSchema:long
   path that is of type http://www.w3.org/2001/
XMLSchema:string
   method that is of type http://www.w3.org/2001/
XMLSchema:string
   target that is of type http://www.w3.org/2001/
XMLSchema:string
   port that is of type http://www.w3.org/2001/XMLSchema:int
Bomb which needs these parameters:
   delay that is of type http://www.w3.org/2001/
XMLSchema:long
Respond which needs these parameters:
   wordcount that is of type http://www.w3.org/2001/
XMLSchema:long
   delay that is of type http://www.w3.org/2001/
XMLSchema:long
snooze which needs these parameters:
   thesecs that is of type http://www.w3.org/2001/
XMLSchema:long
Agent finished.
```

Using the information learned from the WSDL document, we can build a subsequent test agent that calls one of the SOAP methods. For example, we can call the `respond` method by passing a `wordcount` and `delay` value.

To understand WSDL, use a browser to retrieve the WSDL for the example Web service at: http://examples.pushtotest.com:92/axis/servlet/AxisServlet/responder_rpc?wsdl. This is a publicly available SOAP-based Web service hosted by PushToTest.com. Let us see how the test agent learned from the WSDL definition.

```
# Import tells TestMaker where to find Tool objects
from com.pushtotest.tool.parser.wsdl import WSDLParser
from javax.wsdl import WSDLException
```

The agent begins by using import statements to identify the WSDLParser object in TestMaker's TOOL. Details on TOOL are at http://docs.pushtotest.com. WSDLParser provides several methods to discover the contents of the WSDL document.

```
services = definition.getServices()
for keys in services.keySet().iterator():
    current = services.get( keys )
    print "there is a Web Service named", cur-
rent.getQName().getLocalPart(), "offering these services:"
    print
```

The definition.getServices() method returns a services object containing the defined SOAP-based Web services described in the WSDL document. Most WSDL documents define a single Web service. However, the WSDL specification allows for more than one service to be described. The agent uses a *for* loop to iterate through the services.

```
ports = current.getPorts()
    for pkeys in ports.keySet().iterator():
        pcurrent = ports.get( pkeys )
        bind = pcurrent.getBinding()
        print "Service:", pcurrent.getName()
        print "Binding:", bind.getQName()
```

Then, the agent finds the defined ports for each service. A WSDL port describes the transport layer information for the Web service. In this case, the getBinding() method returns a URL to methods needed to bind the agent to the Web service. The Responder is a simple RPC-style service so the binding is simply a URL to call. You can also use the binding information to expose an Applet, client software, or even a security certificate used to call the Web service.

```
        for ops in bind.getBindingOperations():
            if ops.getName()!="class$":
                print ops.getName(), "which needs these \
                parameters:"

                mcur = \
                ops.getOperation().getInput().getMessage()
                if mcur != None:
                    parts = mcur.getParts()
                    for partkeys in \
                    parts.keySet().iterator():
                        part = parts.get( partkeys )
                        print "  ",part.getName(), " \
                        type", part.getTypeName()
```

Next, we use a *for* loop to iterate through the methods and parameters in the Web service.

TestMaker comes with a New Agent Wizard that automatically writes a skeleton of a test agent script. For agents testing SOAP/WSDL-based Web services, the New Agent Wizard reads a WSDL document from a given URL and creates the Jython script that calls TOOL commands necessary to work with the SOAP service. The skeleton is functional and can then be filled out with additional test logic by the user. For details, see the TestMaker documentation.

As we found in the example, TestMaker provides tools to understand the contents of a published WSDL document. From WSDL we learned the available services and methods available to call. Next, we discover how to make requests and receive responses using SOAP.

Constructing SOAP Calls

SOAP is a specification for building interoperating systems. The SOAP specification is broad enough to cover many different types of requests and responses, different styles of parameter encoding, and a variety of service architectures. You need a good knowledge of these SOAP styles to effectively test SOAP-based Web services for scalability, functionality, and performance. The remainder of this chapter shows many SOAP styles in depth and provides a TestMaker test agent as a tangible example.

Different Types of SOAP Parameter Encoding

SOAP is designed to interoperate across platforms, languages, and systems. SOAP implementations deliver interfaces to send a request to a service and pass it parameters that contain characters, numbers, dates, and complex data types. Earlier in this chapter, we delved into interoperability problems that can be introduced by the underlying platform. In this section, we see the different data types that SOAP supports and how software developers can expand on the primitive types to move complex data types in SOAP call.

SOAP defines a set of primitive data types that all implementations must support. These include string, boolean, decimal, float, double, dates and time, binary, and URIs. Each SOAP implementation is responsible for implementing the primitive data types. For example, ASP .NET provides the SOAP data types that are mapped to XML data types in Table 7–1.

Table 7–1 Mapping SOAP Parameter Encoding

ASP.NET data type	Java data type	XML data type
string	String	string
boolean	Boolean	boolean
float(single)	Float	Float
double	Double	double
decimal	Decimal	decimal
long	Long	Long
int	Int	Int
short	Short	short
byte	Byte	unsignedByte
ulong	Ulong	unsignedLong
uint	Uint	unsignedInt
ushort	String	unsignedShort
sbyte	Boolean	byte
DateTime	Float	date
DateTime	Double	Time
DateTime/Date	Decimal	TimeInstant
String[]	Long	ArrayOfString
boolean[]	Int	ArrayOfBoolean

Table 7–1 Mapping SOAP Parameter Encoding (continued)

float[]	Short	ArrayOfFloat
double[]	Byte	ArrayOfDouble
decimal[]	Ulong	ArrayOfDecimal
long[]	Uint	ArrayOfLong
int[]	String	ArrayOfInt
short[]	Boolean	ArrayOfShort

In TestMaker, you can pass a primitive data type using the following format:

```
body.addparameter( "wordcount", java.lang.Long, 1839, None)
```

The `addParameter` method uses the built-in serializer to convert the *long* data value 1839 into an XML representation and sends it to the service. This is easy because of all the underlying Web services code that supports marshaling, encoding, and transmitting primitive data types.

Many times, a transaction needs to communicate complex data types not covered in the primitive data types. Figure 7–5 describes complex data types. In this book, I refer to structured and compound data as complex data types.

SOAP and TestMaker allow custom data types in requests and responses.

```
ef = employeeFile.getFile( 38183 )
body.addparameter( "EmployeeFile", com.pushtotest.employeeFile,
ef, None)
```

In this case, the request to a SOAP service contains the serialized version of an employeeFile object that is referenced by the variable *ef*. TestMaker searches for a registered serializer object that can convert a com.pushtotest.employeeFile object into an XML representation. To register new seri-

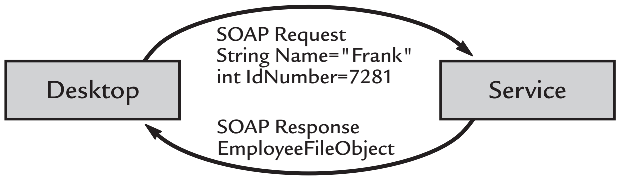


Figure 7–5 SOAP enables complex data types to be used in requests or responses.

alizers with TestMaker, you must write JavaBean objects that respond to serialize and deserialize commands for the specific data type.

The more complex the data, the more serializers are used to handle SOAP requests and responses. For details on using complex data types, see http://docs.pushtotest.com.

Different Types of SOAP Calls

SOAP leverages the XML specification for data types and uses serializer and deserializer objects to marshal and unmarshal data from platform to platform. SOAP also enables several different styles of requests and responses to provide even more flexibility. SOAP enables RPC requests, and document-style requests.

RPC SOAP

RPC-style SOAP calls make a single request to a specific method in an object on a remote server. The request is transported with a list of individual parameters. RPC requests are the closest model to Java and C++ programming models. For example, consider the following object.

```
public class responder_msg
{
    public String Respond( long wordcount, long delay )
    {
       . . .
    }
}
```

responder_msg is a Java object running on a remote server with a single method named Respond that always expects two parameters, both of a *long* data type. The responder_msg object is running now on examples.pushtotest.com, a Web service on the public Internet hosted by PushToTest.com. Next, we examine a TestMaker test agent that interoperates with the responder_msg object using a SOAP request.

```
# test_rpc.a
# Author: fcohen@pushtotest.com

# Import tells TestMaker where to find Tool objects
from com.pushtotest.tool.protocolhandler import \
ProtocolHandler, SOAPProtocol, SOAPBody, SOAPHeader
from com.pushtotest.tool.response import Response
from java.lang import Long

# First we set-up the basic information describing
# the name, location and path to the Web Service host.

host = "examples.pushtotest.com"
port = 92
path = "axis/servlet/AxisServlet"
endpoint = host + ":" + str( port ) + "/" + path

print "SOAP service is at ",endpoint

protocol = ProtocolHandler.getProtocol("soap")
body = SOAPBody()
protocol.setBody(body)

# Set the endpoint values

protocol.setHost( host )
protocol.setPath( path )
protocol.setPort( port )

body.setTarget( "responder_rpc" )
body.setMethod( "Respond" )

# Responder takes these parameters:
# wordcount = the number of jibberish words to return
# in the response XML document. Each word is approximately
# 5-8 characters long and each response is randomly
# unique.
# delay = (optional) the number of milliseconds
#         the Responder service will wait before
#         responding to the request.
# bomb = (optional) the number of milliseconds the
#        Responder service will wait until throwing
#        an exception, which is returned as a 500 Servlet
#        Error and XML fault.
# callback = (optional) see the test_callback.a agent
#            for details

# For this test agent we want 150 jibberish words with
# a 100 millisecond delay

body.addParameter( "wordcount", Long, 150, None )
body.addParameter( "delay", Long, 100, None )
```

```
print "Sending request to server..."

response = protocol.connect()

print
print "Here is the response:"
print response

print
print "Agent done."
```

When you run this agent, TestMaker sends a RPC-style SOAP request to the service. The TestMaker output window displays the response document. Let us look into the agent to see how the RPC-style request is constructed.

```
# Import tells TestMaker where to find Tool objects
from com.pushtotest.tool.protocolhandler import ProtocolHan-
dler, SOAPProtocol, SOAPBody, SOAPHeader
from com.pushtotest.tool.response import Response
from java.lang import Long
```

The `Import` commands tell TestMaker to use the `SOAPProtocol` objects in the TestMaker's TOOL. Normally, we would not have to import a primitive data type such as java.lang.Long since the TestMaker scripting language provides its own long data type. In this case, we need to send a long data type in the parameters of the request.

```
host = "examples.pushtotest.com"
port = 92
path = "axis/servlet/AxisServlet"
```

These variables define the destination server and location of the Web service. To learn more, take a look at the WSDL document for the responder_rpc service at: http://examples.pushtotest.com:92/axis/servlet/AxisServlet/responder_rpc?wsdl

```
protocol = ProtocolHandler.getProtocol("soap")
body = SOAPBody()
protocol.setBody(body)
```

We ask TOOL for a `SOAPProtocol` handler object and SOAPBody object to construct the request. SOAPBody holds the parameters and destination of the request.

```
body.setTarget( "responder_rpc" )
body.setMethod( "Respond" )
```

The target value defines the Web service that holds the object and method we will access. We can get these values from the WSDL document for this Web service too.

```
body.addParameter( "wordcount", Long, 150, None )
body.addParameter( "delay", Long, 100, None )
```

The Responder method takes parameters: `wordcount` is the number of gibberish words to return in the response XML document. Each word is approximately 5–8 characters long and each response is randomly unique; `delay` is an optional parameter containing the number of milliseconds the Responder service will wait before responding to the request; bomb is an optional parameter of the number of milliseconds the Responder service will wait until throwing an exception, which is returned as a 500 Servlet Error and XML fault.

```
response = protocol.connect()
print response
```

Finally, the request is ready to be sent. The `connect()` method sends the request and returns a `response` object.

This simple agent makes a single SOAP request and returns a SOAP response. From this simple agent we could write an agent that does the following:

- Makes multiple requests of various sizes and delays.
- Operates multiple concurrent threads of the request script to increase the load on the server.
- Runs multiple copies of the multithreaded agent on multiple machines.
- Adds simple logging functions to the agents so we can determine TPS and SPI values.

Document-Style SOAP Messages

Document-style SOAP requests work well when an application already has XML data in an object format and does not want or need to convert the object data back into individual parameters, as in the SOAP RPC example in the previous section.

A document-style SOAP-based Web service is running on examples.pushtotest.com, a service hosted on the public Internet by PushToTest.com. This service takes the same parameters as the example RPC-style SOAP request. Instead of taking individual parameters, the document-style SOAP request takes the parameters as elements in a DOM tree.

As an example, let us build a test agent that wants 75 gibberish words with a 25-millisecond delay from the Web service. To do so requires us to build an XML Request document that is sent to the server. The finished document looks like the following:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Body>
  <Respond xmlns="urn:responder_msg">
    <delay xmlns="urn:responder_msg">25</delay>
    <length xmlns="urn:responder_msg">75</length>
  </Respond>
 </s:Body>
</s:Envelope>
```

Below is the test agent in its entirety, followed by a discussion of the steps needed to construct the request and receive a response.

```
# test_rpc.a
# Author: fcohen@pushtotest.com

# Import tells TestMaker where to find Tool objects
from com.pushtotest.tool.protocolhandler import \
ProtocolHandler, SOAPProtocol, SOAPBody, SOAPHeader
from com.pushtotest.tool.response import Response
from java.lang import Long

# This agent also uses JDOM APIs to handle XML data
from org.jdom import Document, Element, JDOMException, \
Namespace, DocType
from org.jdom.output import DOMOutputter, XMLOutputter

# First we set-up the basic information describing
# the name, location and path to the Web Service host.

host = "examples.pushtotest.com"
port = 92
path = "axis/services/responder_msg"
endpoint = host + ":" + str( port ) + "/" + path
```

```
print "SOAP service is at ",endpoint

protocol = ProtocolHandler.getProtocol("soap")
body = SOAPBody()
protocol.setBody(body)

# Set the endpoint values

protocol.setHost( host )
protocol.setPath( path )
protocol.setPort( port )

body.setMethod( "Respond" )

xmlns1 = "urn:responder_msg"

# Create the request document by first creating the Respond
element.
elOne = Element( "Responder", xmlns1 )

# Add child elements for <delay>...
elDelay = Element( "delay", xmlns1 )
elDelay.addContent( "12" )
elOne.addContent( elDelay )

# ... and <length>
elLength = Element( "length", xmlns1 )
elLength.addContent( "25" )
elOne.addContent( elLength )

# then tell the SOAPProtocol handler about the request docu-
ment.
doc = Document( elOne )
body.setDocument( doc )

print "Here is the finished XML request document:"
xo = XMLOutputter()
print xo.outputString( doc )
print

print "Sending request to server..."

response = protocol.connect()
print response
print "Agent done."
```

Following is an example of what we might get back from the response_msg service:

```
<?xml version="1.0" encoding="UTF-8"?>
  <responder>
    <duration>103</duration>
    <message>lingo-text</message>
  </responder>
```

The steps to construct a document-style SOAP request are the same as the RPC-style request agent with one big exception: the script needs to construct a DOM tree representing the XML request data to make the SOAP request. Here is how it works:

```
# Import tells TestMaker where to find Tool objects
from com.pushtotest.tool.protocolhandler import \
ProtocolHandler, SOAPProtocol, SOAPBody, SOAPHeader
from com.pushtotest.tool.response import Response
from java.lang import Long
```

First, we import the `SOAPProtocol` handling objects from the TestMaker TOOL.

```
# This agent also uses JDOM APIs to handle XML data
from org.jdom import Document, Element, JDOMException, \
Namespace, DocType
from org.jdom.output import DOMOutputter, XMLOutputter
```

We also import the JDOM objects to handle construction and parsing of the XML data.

```
host = "examples.pushtotest.com"
port = 92
path = "axis/services/responder_msg"
endpoint = host + ":" + str( port ) + "/" + path
```

These variables define the destination server and location of the Web service. To learn more, take a look at the WSDL document for the responder_msg service at: http://examples.pushtotest.com:92/axis/servlet/AxisServlet/responder_msg?wsdl

```
protocol = ProtocolHandler.getProtocol("soap")
body = SOAPBody()
protocol.setBody(body)
```

We ask TOOL for a `SOAPProtocol` handler object and SOAPBody object to construct the request. SOAPBody holds the parameters and destination of the request.

- Responder takes these parameters in an XML request document: `wordcount` is a long data type that holds the number of gibberish words to return in the response XML document. Each word is approximately 5–8 characters long and each response is pseudo-randomly unique.
- Delay is a long data type that holds the number of milliseconds the Responder service will wait before responding to the request.

```
xmlns1 = "urn:responder_msg"
```

This references the Namespace of the parameters in the request. In document-style SOAP requests, TestMaker uses the Namespace of the first element <Respond> to determine the destination service name. The endpoint gets the request to the right server, the Namespace of the first element gets us to the right Web Service.

We use the JDOM APIs to build the XML request in a DOM object.

```
# Create the request document by first creating the Respond
element.
elOne = Element( "Responder", xmlns1 )

# Add child elements for <delay>...
elDelay = Element( "delay", xmlns1 )
elDelay.addContent( "12" )
elOne.addContent( elDelay )

# ... and <length>
elLength = Element( "length", xmlns1 )
elLength.addContent( "25" )
elOne.addContent( elLength )

# then tell the SOAPProtocol handler about the request document.
doc = Document( elOne )
body.setDocument( doc )
```

Then, we pass the DOM object `doc` to the SOAPBody object to make the SOAP call to the server.

```
response = protocol.connect()
print response
```

Finally, the request is ready to be sent. The `connect()` method sends the request and returns a `response` object. The `Import` commands tell TestMaker to use the `SOAPProtocol` objects in the TestMaker's TOOL.

For those who do not wish to use the JDOM APIs, TOOL provides a convenience method to set the XML document from a String value.

```
myXMLDocument = '<s:Envelope '
myXMLDocument = \
'xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">'
myXMLDocument += '<s:Body>'
myXMLDocument += '<Respond xmlns="urn:responder_msg">'
myXMLDocument += '<delay xmlns="urn:responder_msg">25</
delay>'
myXMLDocument += '<length xmlns="urn:responder_msg">75</
length>'
myXMLDocument += '</Respond>'
myXMLDocument += '</s:Body>'
myXMLDocument += '</s:Envelope>'
body.setDocument( myXMLDocument )
```

In this example the myXMLDocument string contains the XML request to be sent to the host. This can be handy when debugging a SOAP-based Web service. However, this method is also prone to errors since all of the burden on forming a valid XML tree is now in your hands.

This agent is just a start. An expanded agent makes multiple requests of various sizes and delays, operates multiple concurrent threads of the request script to increase the load on the server, runs multiple copies of the multithreaded agent on multiple machines, and adds simple logging functions to the agents so we can determine transactions-per-second (TPS) and SPI values.

Using Formatted XML Data

Consider the times when you already have a formatted XML document. The following agent shows how to send the contents of a file encoded in XML format to a SOAP-based Web service.

```
from com.pushtotest.tool.protocolhandler import ProtocolHan-
dler, SOAPProtocol, SOAPBody, SOAPHeader
from com.pushtotest.tool.response import Response
from java.lang import Long
from org.jdom.input import SAXBuilder
from org.jdom import Namespace, Document

builder = SAXBuilder()
doc = builder.build( 'c:/fileexchange/billing2.xml' )
elEnvelope = doc.getRootElement()
ns=Namespace.getNamespace("env", \
  "http://schemas.xmlsoap.org/soap/envelope/")
  elBody = elEnvelope.getChild( "Body", ns )
  nsm = Namespace.getNamespace("m", \
 "http://www.xerox.com/dcs/billing" )
elSend = elBody.getChild( "send", nsm )
doc2 = Document( elSend.detach() )
print doc2
```

In this unpretentious test agent, we see the `Import` commands being used to access the TestMaker TOOL objects.

```
builder = SAXBuilder()
doc = builder.build( 'c:/fileexchange/billing2.xml' )
elEnvelope = doc.getRootElement()
```

The SAXBuilder object has a special method that takes a file name as a parameter. The builder method reads the contents of the file and returns an XML object containing a DOM tree of the contents of the directory.

```
elBody = elEnvelope.getChild( "Body", ns )
nsm = Namespace.getNamespace("m", \
"http://www.xerox.com/dcs/billing" )
elSend = elBody.getChild( "send", nsm )
doc2 = Document( elSend.detach() )
```

Next, we attach the found XML data in the file to the SOAPBody object. Now we are ready to send the request and receive the response.

Validating Response Data

Validating the response from a SOAP-based Web service highlights the efficiency software developers find when working with XML data. Recall that validating response data requires us to write a script to parse through the HTML codes contained in the response. That parsing is highly error prone and laborious to write and maintain. Here is the equivalent XML validation code to check the response from a SOAP Web service:

```
builder = SAXBuilder()
doc = builder.build( StringReader( resp.toString() ) )
rootElement = doc.getRootElement()
bodyElement = rootElement.getChild( "Body", ns )
for i in bodyElement.getChildren():
    print "Child = ", i
    print "Child name = ", i.getName()
    print "Child content = ", i.getContent()
```

Making It Easier to Write Agents

TestMaker comes with a New Agent Wizard that automatically writes a skeleton of a test agent script. For agents testing SOAP-based Web services, the New Agent Wizard reads a WSDL document from a given URL and creates a TestMaker script that calls the TOOL commands necessary to request the Web service functions. The skeleton is functional and may then be filled out with additional test logic by the user. For details, see the TestMaker documentation.

In addition to the New Agent Wizard, a set of example test agents come with TestMaker. These agents show how to make calls to SOAP-based Web services. The example agents may also serve as an outline for your own test agents. Feel free to start with one of the example agents as a template.

Lastly, while testing a SOAP-based Web service, seeing the HTTP protocol data as it moves across the network connection between the browser and the server can be useful to build and debug intelligent test agents. Windows NT, 2000, and XP users may view network traffic with the Microsoft Network Monitor utility. This utility captures the packets of data and displays them to show the `GET` and `POST` commands and their associated data values. Linux users may view network traffic with the popular tcpdump (http://www.tcpdump.org), ethereal (http://www.ethereal.com/), and NetWatch (http://www.slctech.org/~mackay/netwatch.html) utilities.

Resources

For further information about the SOAP and WSDL standards, go to:

- WSDL, http://www.w3.org/TR/ws-desc-reqs/
- SOAP, http://www.w3.org/TR/soap12-part0/
- XML-RPC, http://www.xmlrpc.com/
- Web Services in general, http://www.w3.org/2002/ws/

Summary

This chapter covered the basics of SOAP and WSDL technologies. We learned how TestMaker provides scripting language commands and a library of test objects to work with WSDL documents and many styles and forms of SOAP requests. Finally, this chapter showed how much simpler it is to parse through an XML response than parsing through raw HTML.

In the next chapter, we see how intelligent test agents are used to set up tests, including how a TestMaker agent reads directly from a database.

8
State, Design, and Testing

The examples in past chapters assume that each request to a Web-enabled application is stateless. In a stateless system, each request is separate from the others. Of course, we now know from using Web-enabled applications that many times stateful requests are important, if not outright necessary. This chapter explores why state is important to Web-enabled applications, discusses a methodology for testing stateful systems, and shows a practical example of how to configure a stateful system to be tested.

A Question of State

Web technology introduces a collection of protocols, providers, and languages. Perhaps the greatest feature of these is the stateless nature of Web protocols. In a stateless system each request and response is atomic—none depend on previous or subsequent requests or responses. The request comes in to a service and a response goes out and that is all there is to a transaction. The benefits to stateful systems include:

- Capacity planning is easy. For example, measuring performance at 100 and 500 concurrent users tells how many servers will be needed when 1,000 and 10,000 users show up.
- Content caching is more plausible. In my experience, stateless systems work well for publishing content that seldom changes.

The more static the content, the more efficient distributed cache servers may deliver content to users.

- Distributed datacenters more equally share the load of a population of users since it does not matter which datacenter handles a request. This has the added advantage of providing greater stability to the overall system.

With the maturation of Web technology, today we find many places where stateful technology is necessary to build Web-enabled applications. Consider a system that requires a user to sign in to receive a personalized or confidential response from the server. In a stateless Web-enabled application, the user would fill in their id and password on each and every Web page! Many times that is just not practical—and it is contrary to the way the majority of Web users think the Web works.

By adding a little statefulness—for example, a cookie value to track a sign-in session—the user experience is made easier and the underlying protocols are still scalable. Unfortunately, the situation is like Pandora's box. Now that the stateful box is open, software developers, QA technicians, and IT managers are having to work their way through a huge volume of issues, problems, and challenges related to stateful systems. Here are a few of my favorites:

- Cookies—Provide a simple way to identify sessions among a group of HTTP/HTML requests. The cookie value is often an index into a table stored in the memory of a Web server that points to an in-memory object holding the user's records. This has many potential problems: If the user's request is routed to a different server in a subsequent request, the session information is unknown to the server. If the user is routed to a different server and the server is part of an application cluster, then all the servers that could receive the user's request must have a way to synchronize the session data. Storing cookies and synchronizing sessions among clusters of servers usually requires configuration, storage space, and memory.
- URL rewriting—Instead of storing a cookie value in the HTTP header of a request, the URL is rewritten to include a session parameter. URL rewriting might avoid cookies but it shares the same set of potential problems just mentioned above. Plus, with URL rewriting there are no static URLs in your Web-enabled application, which often makes caching and indexing more

difficult. Finally, every Web page needs to be dynamically generated so all hyperlinks include the session parameter.

- SAML Artifacts—SAML is a security protocol that enables Web-enabled applications to authenticate users and data. SAML was largely designed for Web-enabled applications where browser interfaces over HTTPS/HTML provide an interface to security functions, but it also has good use in Web services. SAML uses an Artifact when it needs to share data that is too large to fit into the parameters of a URL. By design, the Artifact needs to be stored so that it may be later retrieved and used. This requires software configuration for Artifacts, memory, and storage. See my article about SAML at: http://www-106.ibm.com/developerworks/xml/library/x-samlmyth.html

The more state we design into Web-enabled applications, the more workarounds we will need to develop to use the stateless nature of Web technology. These workarounds often look like chaos embodied to software developers, QA technicians, and IT managers. When testing a stateful system, we need to look in two dimensions at system performance under increasing loads of use:

- **Vertical scalability** enables the system to handle individual transactions in a timely manner.
- **Horizontal scalability** is the ability of the system to handle all the requests concurrently.

For this discussion, imagine a transaction that spans several steps. For example, in a transaction a user signs in to a Web-enabled application, reads the first waiting message, deletes the message, and checks the rest of the waiting messages. Figure 8–1 illustrates this example. We run the same transaction two times. When I compare the two transaction times, I ask these questions:

- Is the overall performance time acceptable to the user's goals?
- Which parts of the transaction varied in performance?
- How much did they vary from test to test?
- Was I expecting a variation? If so, was it the variation I expected?
- Does the variation appear to impact the performance of other parts of the transaction?

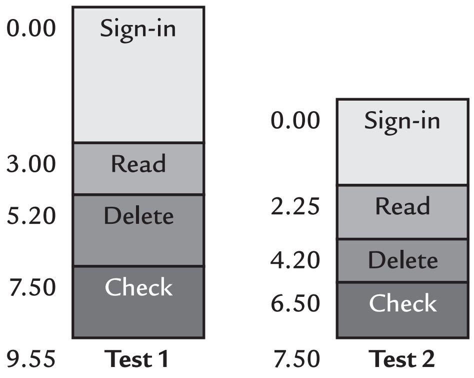


Figure 8–1 Systems scale in two directions: vertically to improve the ability of the service to respond to each request and horizontally to improve the ability of the service to handle increasing quantities of requests. This figure shows a vertical scalability test where two runs of the same transaction result in different response times.

In my experience, the same set of questions apply to all of the Web-enabled applications I design and test. By instrumenting the software and observing the system performance through a test agent log, I can see where the system needs optimization. Later, once system changes are in place, the same test proofs the improvements.

Looking at a group of individual request and response transactions like those in Figure 8–1 show that some transactions are handled rapidly and others take longer. The ideal vertically scalable system has a constant proportion of fast responses to slow responses. Vertical scalability problems appear when more requests result in transactions that on average take increasing amounts of time to process.

Horizontal scalability is the ability of the system to handle all requests concurrently. Figure 8–2 shows an example of horizontal scalability being tested. Servers need resources to respond to requests. Eventually the resources are exhausted and the server has to refuse additional connections and requests. Load balancers, cluster technology, and multicasting services are proven solutions that enable groups of servers to act as one system to provide horizontal scalability.

In my view, stretching Web technology so far toward statefulness is a leading cause for poor performance and scalability problems. And the more state a system incorporates, the more complex becomes system configuration in advance of a test. For example, consider the stateful system that only exhibits

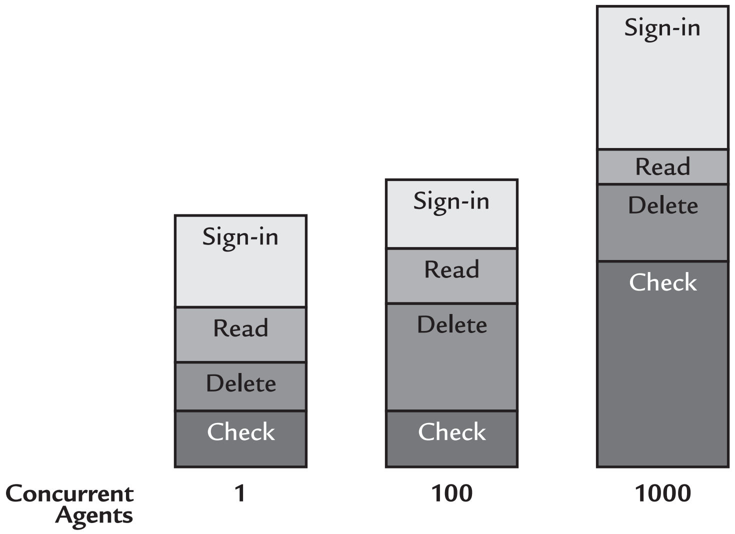


Figure 8–2 Horizontal scalability tests a system's ability to handle increasing levels of load. Notice that the parts of a transaction perform differently under different levels of load.

performance problems after a certain amount of usage. This is the example I look into closely in the next section.

Lifecycle for Testing Stateful Systems

Developing intelligent test agents very much resembles developing software applications, including the need for a repeatable lifecycle process. Testing systems requires a certain amount of preparation, setup, execution, and analysis. The better-designed tests are designed around a lifecycle that automates these steps from configuration all the way to reporting results. Figure 8–3 shows a summary of the test lifecycle.

In the preparation stage of a test, the test designer collects a list of steps needed to configure the system to a known state prior to testing. The goal here is not to configure the system (that is coming in the set-up stage next), but to understand at an abstract level the steps needed to make the system ready to test. For example, when testing an ecommerce system that shows business telephone customers discount phone service plans, the system may need to retrieve a list of current service plans to emulate the actions of a cus-

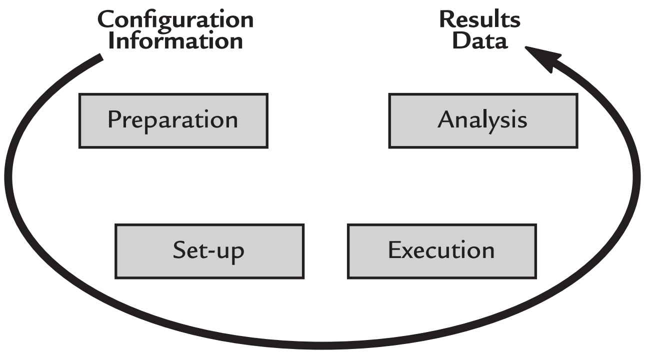


Figure 8–3 The lifecycle for testing stateful systems goes from configuration of the system through a lifecycle of steps to reporting results.

tomer. The preparation stage would identify the commands needed to get the list of current service plans.

The advantage of having a preparation stage is that tests become more dynamic and automatic. By abstracting the actions that will be taken during subsequent setup and execution phases, should different preparation be needed, the setup, execution, and analysis portions of the test agent do not need to be changed. Additionally, the abstracted actions during preparation mean that any agent, script, or program can follow the actions to configure a system for a test automatically.

The next two steps—setup and execution—do the laborious tasks of performing the test. Setup performs all the tasks defined in the preparation stage, including communicating with Web-enabled applications, databases, files, and message delivery systems to configure the target test system. Setup also performs validation to check that the actual preparation of the system succeeded. The setup step may even veto the overall test if the defined test parameters for configuration are not achieved.

The execution stage is responsible for performing the test steps in the order and frequency defined by the test designer. The execution stage receives run-time parameters for the test from the user through a console or from the setup stage through configuration files. The execution stage's output is data in a format defined for the analysis stage. For simple tests the output may appear in a window on the computer console. More advanced test agents may create delimited data files or store the results data in a relational database for the analysis stage's consumption.

In the analysis stage, the information derived from the execution of a test is turned to actionable knowledge. The analysis stage begins with a collection of raw results data. The data becomes valuable when management learns the quantitative metrics of the system's quality and performance. The results also benefit engineers by giving them insight into their code that was not available when the same code was run on a development system. As one manager on the SunONE Application Server team said: "We need a taste of the latest build's performance." The analysis stage ends with the raw data compiled into a concise report that can then be used to present the results for action to management.

In the next section, we learn how to apply the lifecycle to an example scalability and performance study.

Techniques to Establish State

Consider the case of a content publishing system that features threaded discussions. On a daily basis many messages are read, several new discussions are started, and subscribers post many `reply` messages. The company hosting the publishing system needs to plan for an expansion in usage that is anticipated as the result of new marketing projects. Management needs to know the existing datacenter's capacity and plan the budget to buy the incremental equipment needed to handle the new users. Figure 8–4 shows the nature of the discussion system. Discussions are grouped by topic, with each

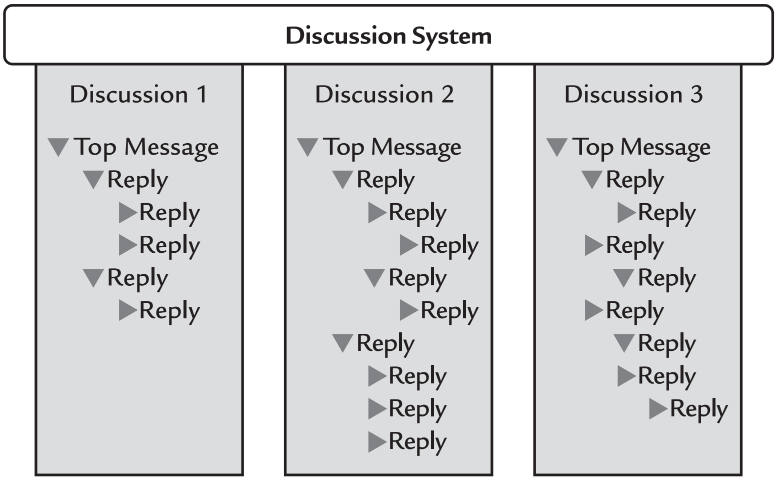


Figure 8–4 An example discussion system hosts multiple threaded discussions.

discussion containing a top message and threaded `reply` messages. The Server Side, a popular Web site for J2EE developers, hosts a discussion system of this nature at http://www.theserverside.com.

If a scale of stateless to stateful systems exists, we would find this example system much toward the stateful side of the scale. Consider the state issues needed to operate a scalability and performance test:

- Users must be identified to have posting privileges to create a new discussion, top message, or `reply` message.
- To create a `reply` message, the user must first browse either a top message or `reply` message. Browsing these messages sets the context for the `reply` message.
- Messages appear showing its immediate `reply` messages. The user needs to walk through a list of messages to see all the replies.

Testing the discussion system raises several classical problems with testing in general. The preface to this test is that an existing system needs to be evaluated for performance under increasing levels of load. Tests of stateful systems, by their nature, change the system being tested. So throwing a few thousand concurrently running intelligent test agents at the existing system may cause the entire system to go down. While this would be helpful for the company to know such a flaw exists, it is also highly disruptive to the existing users. Let's consider some of the possible environments in testing existing stateful systems:

- Take a snapshot of the current discussion software state and install the snapshot on a duplicate datacenter. Here the test would start with a baseline that matches the current system state. By testing on a duplicate datacenter the existing users are not impacted. There are two drawbacks to such a test. First, duplicating a datacenter for a test may not be practical because of the expense and time it takes to undertake such an effort. Second, conducting the test on the real datacenter may uncover problems that do not appear on a duplicate system—creating an *exact* duplicate can become a lifelong quest.
- Test with the production system off-line. Instruct the existing users that tests will be conducted during certain off-peak usage

hours. During those hours, after making a system backup, conduct tests on freshly installed application software. After the tests, the system is restored from the backup. The downside to this test is that the test environment may not be the same as the real production environment. Backing up and restoring the environment may mask problems. For example, the effects of a system with memory effects would not be tested because the backup and restore operation removes memory leaks.

- Continuous testing—Test while the system is up and running but partition the system so that test data is not available to existing users. In the discussion system example, the test creates a new discussion group that is visible only to the test agent and not to the existing users. The test agent posts messages to the limited visibility discussion group and measurements of the effects of the test activity are shown as the incremental activity over the normal activity from the existing users.

Regardless of the test scenario chosen, the tests will be run several times and require configuration to the same state each time. Using a repeatable preparation and setup technique improves the speed at which a test may be conducted and minimizes the risk that the initial state during each test run varies, which usually creates a variance in the results. Figure 8–5 shows a technique for establishing state in a system to be tested.

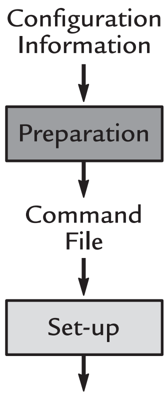


Figure 8–5 A command protocol is shared between the preparation and setup stages to establish state prior to testing a stateful system.

In the discussion system example, the setup stage to configure the system for a test has these possible commands:

- Reset the entire system
- Delete a discussion group
- Start a new discussion group
- Create a top message
- Reply to a message

These form the basis of a simple command protocol that the setup stage of the test can process to prepare the system for a test.

In the discussion system example, we assign command codes to each configuration command:

Command	Task
1	Reset the entire system
2	Delete a discussion group
3	Start a new discussion group
4	Create a top message
5	Reply to a message

With this command protocol we could configure a new discussion group by processing the following instructions:

```
1, 3, 4, 5
```

When processed one after the next, these commands configure the state of the system to be tested.

The preparation and setup technique offers a lot of flexibility in the source of the configuration information. Figure 8–6 shows the technique reading from a file, database, and even a Web-enabled application as the source of the configuration instructions to establish state in a system to be tested.

In the next section we put the preparation and setup technique into practice using intelligent test agents.

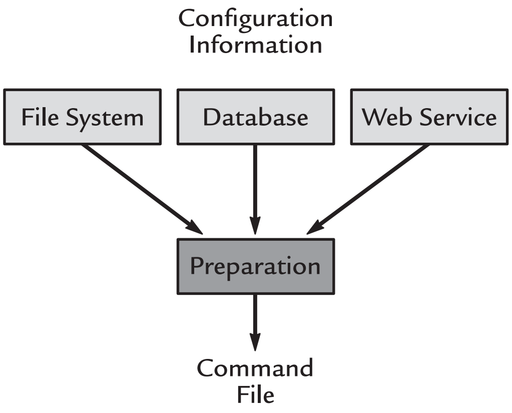


Figure 8–6 The preparation stage of a test takes input from several sources, including preparation instructions stored in file systems, databases, and even Web-enabled applications, and produces a command file that the setup stage uses.

Preparation and Setup Agents

Next, we examine a TestMaker test agent that performs the preparation stage of configuring the example discussion system for a test. Following is the agent in its entirety and an explanation of how it works. All of the code presented in this book is also available for download at http://www.pushtotest.com/ptt/thebook.html.

```
# Prepare_setup_agent.a
# Author: fcohen@pushtotest.com

print "Agent running: Prepare_Setup_agent.a"
print "Description:"
print " This is the first of two agents that"
print " demonstration how to automate a Web-enabled"
print "application test:"
print
print "  1) Prepare_setup_agent.a - creates a "
print "     delimited data file containing"
print "     instructions for the setup_agent.a "
print "     agent to follow."
print "  2) Setup_agent.a - issues commands to an"
print "     example ecollaboration"
print "     Web-enabled application to create the"
print "     needed state to run the test."
```

```
print

# Parameter settings

# The file to store the commands

filename = "d://agents//experimental//commands"


# The host to set-up

dest_url = "examples.pushtotest.com"
dest_params = "responder"


# Import tells TestMaker where to find TOOL objects

from com.pushtotest.tool.util import Lingo
import random
import sys


# A mini-function that writes the command to the file and
# handles exceptions. The file contains instructions for
# the setup_agent to follow. Setup_commands uses the
# following format:
# command-type, url, parameters, message_size

def write_a_command( file, command ):

    # The global command makes tells the function to use
    # variables defined outside of this function. Otherwise
    # this function would define it's own dest_url variable
    # locally

    global dest_url, dest_params, tick

    thecommand = str( command ) + "," + dest_url + "," + \
    dest_params + "," + str( random.randrange(100,150) ) \
    + "\n"
    print "writing this command:",thecommand

    try:
        file.write( thecommand )
    except Exception, ex:
        print "Uh, oh:",ex
        print "Closing file and exiting."
        file.close()
        sys.exit

    tick += 1



# Main:
# Start by first creating a data file for the commands

fn = filename + ".db"
print "Building:", fn

# The w means write, which creates a new file, deleting
# any existing file with the same name. One could also
# use a for append, which appends entries to the end of
# an existing file. r opens an existing file to be read.

f1 = open( fn, "w")

tick = 0    # This will keep a count of the commands we write
print


# Define the constants used to create the command file.
# A copy of these constants will be found in
# Setup_agent.a too.

new_thread = 0
new_node = 1
pop = 2
reply = 3

# Write the configuration commands to initialize a new
# discussion group to a known state.

write_a_command( f1, new_thread )
write_a_command( f1, new_node )
write_a_command( f1, reply )
write_a_command( f1, pop )
write_a_command( f1, new_node )
write_a_command( f1, new_node )
write_a_command( f1, reply )
write_a_command( f1, reply )
write_a_command( f1, pop )
write_a_command( f1, pop )
write_a_command( f1, new_node )

print
print "Wrote", tick, "commands in total."
# Close the file

print
print "Closing the file."
f1.close()


# Run the Setup_agent.a and then return back to this agent

print
print "Running the Setup_agent.a agent."

execfile( "d://agents//thebook//chapter5-setup//
Setup_agent.a" )

print
print "Agent done."
```

The `Prepare_Setup_Agent` is the first of two agents that demonstrate how to automatically configure a stateful system for a test. This agent creates a delimited format data file containing instructions for `Setup_agent` to use to configure the discussion system.

`Prepare_Setup_Agent` begins by establishing a variable to define the path and file name for the setup agent's command file. This is the file where the `Prepare_Setup_Agent` agent stores configuration commands.

```
filename = "d://agents//experimental//commands"
```

Note in this example a Windows-style path and file is used. The agent could just as easily use a Unix-style path such as:

```
filename = "/home/fcohen/commands"
```

The agent then imports the sys object that it will use later to exit the agent in case of an error or exceptional state.

```
import sys
```

The `write_a_command()` function is a small utility function that writes a configuration command to the file and handles file-related exceptions. The file contains instructions for the `Setup_agent` to follow and uses the following format: command-type, URL, parameters, message_size. Comma characters separate the individual fields and each record ends with a carriage return character.

```
def write_a_command( file, command ):
```

The global command tells the `write_a_command()` function to use the variables defined outside of this function. Otherwise, this function would define and use its own `dest_url`, `dest_params`, and `tick` variables locally.

```
    global dest_url, dest_params, tick
```

The `write_a_command()` function accepts a reference to an already open file handler (file). The next command builds a command and then writes it to the file.

```
    thecommand = str( command ) + "," + dest_url + "," +
dest_params + "," + str( random.randrange(100,150) ) + "\n"
```

The `tick` variable keeps track of a count of commands written to the file. Here we use the shorthandle += to indicate the value of `tick` is replaced with the current value plus one.

```
    tick += 1
```

Next comes the main body of the agent. The agent opens the file using the "w" option to create a new file, including deleting any existing file with the same name. The agent could also use the "a" option for append, which appends entries to the end of an existing file. The "r" option opens an existing file to be read.

```
f1 = open( fn, "w")
```

The `Prepare_Setup_Agent` agent defines a simple command protocol that the `Setup_agent` will use to configure the discussion system. The agent defines constants to identify commands that are added to the *command* file. A copy of these constants is also in the `Setup_agent`. The constants implement the following test commands:

- `new_thread`—Starts a new thread of discussion and posts the top-most message.
- `new_node`—Reply to the last message posted, this remembers the current message location for a subsequent `pop` command.
- `pop`—Replies to the message created with previous `new_node` command.
- `reply`—Replies to the last node posted.

The `Prepare_Setup_Agent` creates a discussion group, illustrated in Figure 8–7, by using a combination of these commands.

The agent writes commands to the configuration command file to create the discussion illustrated in Figure 8–7:

- `new_thread`—Creates 1
- `new_node`—Creates 2
- `reply`—Creates 3
- `pop`—Moves back to 1
- `new_node`—Creates 4
- `new_node`—Creates 5
- `reply`—Creates 6
- `reply`—Creates 7
- `pop`—Moves back to 4
- `pop`—Moves back to 1
- `new_node`—Creates 8

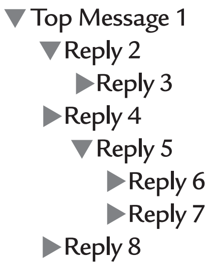


Figure 8–7 The desired final state of a new discussion group is defined in the preparation configuration file.

The agent then closes the command file. Without closing the file, the `Setup_agent` would encounter a file input/output error when trying to open the command file.

```
f1.close()
```

Lastly, the agent runs `Setup_agent` using the `execfile` command. `exec-file` runs an agent and returns back once `Setup_agent` finishes. `execfile` simplifies the task of writing a master agent that runs agents.

```
execfile( "d://agents//thebook//chapter5- \
-setup//Setup_agent.a" )
```

Setup_agent

The previous section showed how `Prepare_Setup_Agent` creates the *commands* file that contains configuration commands to establish the correct state in the example discussion system. Following is `Setup_agent` in its entirety with an explanation of how it works.

```
# Setup_agent.a
# Created on: June 30, 2002
# Author: fcohen@pushtotest.com

print "Agent running: Setup_agent.a"
print "Description:"
print "  This the second of two agents that "
print "  demonstration how to"
print "  automate a Web-enabled application test:"
print
print "  1) Prepare_setup_agent.a - creates a delimited"
print "     data file containing"
print "     instructions for the setup_agent.a agent"
print "     to follow."
print "  2) Setup_agent.a - issues commands to an"
print "     example ecollaboration Web-enabled application"
print "     to create the needed state to run the test."
print


# Parameter settings

# The file to store the commands
```

```
commandfilename = "d://agents//experimental//commands"


# The host to set-up

dest_url = "examples.pushtotest.com"
dest_params = "responder"


host = "examples.pushtotest.com"
port = 92
path = "axis/servlet/AxisServlet"
endpoint = host + ":" + str( port ) + "/" + path


# Import the sys object to use the exit() method for errors
import sys


# Post a message function
# input the parent_message number
#
# Note: This is just a simulation of a Web-enabled applica-
tion request.
#       Someday I may get around to coding a real service.

def post_message( msg_num, theindent, type, url, params ):
    print indent*'   ', type, msg_num, ":", url, params


# Simple Stack implementation
# Used to keep track of nodes in the message hierarchy

def push( value ):
    global stack
    stack += [ value ]

def popone():
    global stack
    return stack.pop()


# Main:

# Let's start by first opening the command data file
```

```
try:
    target =  commandfilename + ".db"
    print "Opening file: ", target
    print

    dbfile = open( target )

except IOError, e:
    print "Error while opening the file:"
    print e
    print
    sys.exit


# Define the constants used to create the command file.
# Prepare_setup.a uses the same constants when it builds
# the command file.

# Starts a new thread of discussion and posts
# the top-most message
new_thread = 0

# Reply to the last message posted, this remembers the
# current message location for a subsequent pop command
new_node = 1

# pop replies to the message created with previous
# new_node command
pop = 2

# Replies to the last node posted
reply = 3

# So if we wanted to create a message tree like this:
# top-message 1
#   reply 2
#     reply 3
#   reply 4
#     reply 5
#       reply 6
#       reply 7
#   reply 8

# we could use the following commands:
# new_thread - creates 1
# new_node - creates 2
# reply - creates 3
# pop - moves back to 1
# new_node - creates 4
# new_node - creates 5
# reply - creates 6
# reply - creates 7
# pop - moves back to 4
# pop - moves back to 1
# new_node - creates 8

# And that's just what Prepare_setup.a did to build the
# command.db file. So we will now loop through each record
# in the command.db file and process each command.


# Create the variables

tick = 0            # Counter for the number of commands pro-
cessed
next_message = 0    # Identifier for the next new message
indent = 0          # Used to figure out the indentation of a
reply
stack = []          # used to build the tree

# Process the commands

for line in dbfile.readlines():
    commands = line.split(",")
    command = commands[0]
    handled = 0

    if int( command ) == new_thread:
        handled=1
        tick += 1
        node=post_message(next_message,indent, \
          "top-message",commands[1],commands[2])
        push( indent )
        next_message += 1
        indent += 1

    if int( command ) == new_node:
        handled=1
        tick += 1
        node=post_message(next_message,indent, \
       "node-reply",commands[1],commands[2])
        push( indent )
        next_message += 1
        indent += 1
    if int( command ) == reply:
        handled=1
        tick += 1
        node = post_message(next_message, indent, \
        "reply", commands[1], commands[2] )
        next_message += 1

    if int( command ) == pop:
        handled=1
        tick += 1
        indent = popone()

    if handled==0:
        print "Unknown command!"
        print line


# Cleaning up

print
print "Processed", tick, "commands in total."

print
print "Closing the file."
dbfile.close()

print
print "The collaborative Web-enabled application is cor-
rectly set-up"
print "and ready to test."
print
print "Agent done."
```

The `Setup_agent` reads from the *commands* file and executes the commands one at a time. `Setup_agent` issues Web-enabled application requests to the example discussion system to build a new discussion group that is populated with a tree of `reply` messages.

```
def post_message( msg_num, theindent, type, url, params ):
```

While the `post_message()` function does not actually send a request to a Web-enabled application, it does print the message to the TestMaker output window, including printing the correct indentation. The `indent*'    '` command multiplies the string of four space characters by the indent variable value. If indent equals 2, then 8 space characters are output.

```
    print indent*'    ', type, msg_num, ":", url, params
```

The *commands* file created by `Prepare_Setup_Agent` in the previous section contains a list of commands. The *commands* file uses the `push` and `pop` commands to keep track of when to conclude adding reply messages to a "branch" of replies. Figure 8–8 illustrates the branch locations. For example, after writing Reply 7, the *commands* file issues `pop` commands until we are back at the Top Message 1 and then it commands `Setup_agent` to write Reply 8.

`Setup_agent` implements a simple `push/pop` stack to keep track of the branches in the hierarchy of a discussion group. `push()` puts the current branch number onto a stack of values, one value for each branch.

```
def push( value ):
    global stack
    stack += [ value ]
```

The `popone()` function pulls the most recently pushed branch number off of the top of the stack.

```
def popone():
    global stack
    return stack.pop()
```

The `push/pop` stack may be applied to your other agents. To see the stack in operation one could add the following commands to print the contents of the stack as items are pushed and popped:

```
for stack_item in stack:
    print stack_item
```

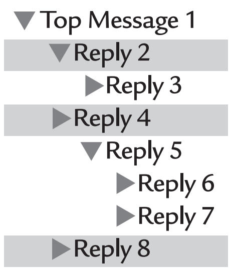


Figure 8–8 Reply messages 2, 4, and 8 are branches off the Top Message. While the discussion group is being created, the agent must remember the branch locations to know how to handle commands to create new branches.

The agent begins the main body of the commands by opening the *commands* file.

```
dbfile = open( target )
```

The default open command opens a file for read access.

The agent then duplicates the command constants originally defined in `Prepare_Setup_Agent`. The agent loops through each record in the command.db file and processes each command. While the `post_message()` function does not actually send a request to a Web-enabled application, it does print the `reply` message with the correct indentation to the screen.

The agent then creates variables used during command processing, including a counter for the number of commands processed, an identifier for the next new message, and an integer value to track the indentation of the next `reply` message in the tree of `reply` messages.

`Setup_agent` processes each command in the *commands* file. The `for` command loops through every line in the *commands* file.

```
for line in dbfile.readlines():
```

The agent creates a list variable called `commands` by reading a line from the *commands* file and splitting it into a list of individual values by delimiting each value by a comma character using the split function.

```
    commands = line.split(",")
```

The `commands` variable is a list so `commands[0]` refers to the first value found in the current line of the *commands* file. In the discussion system example, the first time through this loop `commands[0]` contains a `new_thread` command.

The agent handles commands and takes action accordingly. When handling a `new_thread` command, the agent indicates the command has been handled, bumps up the number of handled commands, posts the message using the `post_message()` function, pushes the indentation value onto the stack using the `push()` function, then bumps up the number of messages and the indentation value.

```
    if int( command ) == new_thread:
        handled=1
        tick += 1
        node=post_message(next_message,indent, \
        "top-message",commands[1],commands[2])
        push( indent )
        next_message += 1
        indent += 1
```

The handled variable is a simple flag that the agent uses to make sure every command is recognized and executed. If handled is still 0 at the end of the next block of code, then an error is thrown.

```
        if handled==0:
            print "Unknown command!"
            print line
```

`Prepare_Setup_Agent` automatically runs `Setup_agent` using the `execfile` command. When `execfile` is used, `Setup_agent` exits back to `Prepare_Setup_Agent` then execution continues after the `execfile` command.

Using Databases to Configure Tests

Earlier in this chapter we learned how the preparation stage of a test formats configuration instructions into a command file. The configuration instructions may be manually coded into a preparation test agent retrieved from a file or database. This section shows how an intelligent test agent that was built in TestMaker interacts with relational database systems.

TestMaker provides a rich environment for building intelligent test agents, but at its heart TestMaker is just another application written in Java. The Java language provides database connectivity through the JDBC API. JDBC drivers are available for most commercial and open source databases, including Oracle, Microsoft SQL Server, and PostgreSQL.

Following is a test agent that demonstrates how to use JDBC functions from within a test agent environment.

```
# DB_meister.a
# Created on: July 3, 2002
# Author: fcohen@pushtotest.com

print "Agent running: DB_meister.a"
print "Description:"
print "  Shows how an agent may use a relational "
print "  database to receive or record data."
print
```

```
print "  NOTE: THIS AGENT DOES NOT ACTUALLY RUN since "
print "  there is no relational database source"
print "  available and this agent print depends on "
print "  a JDBC driver that you must supply."
print


# Please configure this agent and then remove the
# sys.exit command

print
print "Stopping agent: Please configure this agent to your"
print "own JDBC driver and relational database source."
print
sys.exit


# Import tells TestMaker where to find TOOL, JDBC objects
import sys
from java.sql import DriverManager


# JDBC Driver information

# Note: You must provide these parameters to
# use your provided JDBC driver. You must also add the JDBC
# driver code to the TestMaker classpath. Set the classpath
# by changing the testmaker/testmaker/bin/ide.cfg file.

JDBCDriverClassName = "com.ashna.jturbo.driver.Driver"
jdbcURL = "jdbc:JTurbo://myrelationaldatabase:1410:MSSQL"
jdbcUserID = "myuserid"
jdbcPassword = "thepassword"


# Load the JDBC driver

try:
    java.lang.Class.forName( JDBCDriverClassName )
    myConn = DriverManager.getConnection( jdbcURL, jdbcUse-
rID, jdbcPassword )
except:
    print
    print "Could not load JDBC driver: ", \
    JDBCDriverClassName
    sys.exit
```

```
# Show example of reading from a relational database

print
print "=================================================="
print "Information from the Customers table:"

# Customer count

query1 = "select count(*) from Customers"

try:
statement = myConn.createStatement()
results = statement.executeQuery( query1 )
results.next()
except Exception, ex:
print "Exception while counting customers: ",ex
sys.exit

print
print "  Customer count: ", results.int



# Customer list

query2 = "select customer_number, last_name, "
query2 += "salutation, city, year"
query2 += " from table"
query2 += " order by last_name"

try:
statement2 = myConn.createStatement()
r2 = statement.executeQuery( query2 )
except Exception, ex:
print "Exception while getting customer list: ",ex
sys.exit

print
print "  Complete customer list (sorted by last name):"

while results2.next():
print " ",r2.getString,r2.getString,r2.getString, \
    r2.getString(4),r2.getString(5)


#----------------------------------------------------
```

```
# Inserting data into a table

insert1 = "insert into customers files (id, name,"
insert1 += "manager, phone) "
insert1 += "VALUES (1001, 'Simpson', 'Mr.', "
insert1 += "'Springfield', 2001)"

try:
i1 = myConn.createStatement();
ir1 = s2.executeQuery( insert1 );
except Exception, ex:
print "Exception while inserting a record: ",ex


print
print "Agent ended."
```

Let us look at the agent script's individual sections to learn what the script does to communicate with a relational database. Unlike most of the other test agents presented in this book, the `DB_meister` agent does not actually run since at the time of writing this book there was no publicly available relational database source for the agent to connect to. Check the http://examples.pushtotest.com Web site to see if an example database source is now available.

```
sys.exit
```

By default this agent starts and then exits. To run this agent, please configure the agent to connect to an available database and then remove the sys.exit command.

The `Import` command tells TestMaker where to find the standard Java libraries that implement an interface to the JDBC driver.

```
from java.sql import DriverManager
```

A JDBC driver implements all the code needed to communicate with a specific type of database and an interface that conforms to the JDBC specification published by Sun for the Java platform. Database providers and third-party driver manufacturers provide JDBC drivers. A driver is packaged as a JAR file. Adding the driver JAR file to the TestMaker classpath is required to give test agents access to the driver's functions. For instructions on adding resources to the classpath, see the TestMaker documentation.

The test agent sets four basic parameters to tell the JDBC driver where and how to connect to the database. `JDBCDriverClassName` points to the driver object in a Java namespace defined by the driver manufacturer. The `jdbcURL` parameter describes which JDBC driver to use (`JTurbo`), the URL to the database, and the database name (`ORCL`.)

```
JDBCDriverClassName = "com.ashna.jturbo.driver.Driver"
jdbcURL = "jdbc:JTurbo://myrelationaldatabase:1527:ORCL"
```

Driver and database manufacturers define the format to the `JDBCDriver-ClassName` and `jdbcURL` parameters. Here is a short list of common JDBC drivers and their parameters:

For IBM DB2 (http://www.ibm.com), the agents use:

```
JDBCDriverClassName = "COM.ibm.db2.jdbc.net.DB2Driver"
jdbcURL = "jdbc:db2://myrelationaldatabase.com/test"
```

For Oracle (http://www.oracle.com), agents use:

```
JDBCDriverClassName = "oracle.jdbc.driver.OracleDriver"
jdbcURL = "jdbc:oracle:thin:@myrelationaldata-
base.com:1527:ORCL
```

JTurbo is a JDBC driver for MS SQL server (http://www.newatlanta.com/index.jsp):

```
JDBCDriverClassName = "com.ashna.jturbo.driver.Driver"
jdbcURL = "jdbc:JTurbo://myrelationaldatabase.com:1433/ \
dbhost/sql70=true"
```

JDBC drivers use Java's class loader functions to find and access a driver's functions. Note that the agent code uses the `try/except` format so as to trap an error that may happen when the driver loads. Possible errors include Java not finding the driver, the driver not being able to connect to the database, or the wrong configuration information.

```
try:
    java.lang.Class.forName( JDBCDriverClassName )
    myConn = DriverManager.getConnection( jdbcURL, \
    jdbcUserID, jdbcPassword )
except:
    print
    print "Could not load driver: ", JDBCDriverClassName
    sys.exit
```

Now that the driver is loaded, the agent shows a few SQL functions to retrieve data from the Customers table in the database. The Customers table has these fields and descriptions:

- `customer_number long`
- `last_name char(50)`
- `salutation char(10)`
- `city char(50)`
- `year char(10)`

The first query counts the number of customers in the Customers table using a SQL select statement.

```
query1 = "select count(*) from Customers"
```

Executing the query requires a connection object and statement object. The agent uses the `try/except` format to catch errors that happen while processing the database function.

```
try:
statement = myConn.createStatement()
results = statement.executeQuery( query1 )
results.next()
except Exception, ex:
print "Exception while counting customers: ",ex
sys.exit
```

The query result is returned in the results object. Using the `next()` method retrieves the first row of the database results. In this case there is only a single row, which is formatted and printed to the TestMaker output window.

```
print "  Customer count: ", results.int
```

The second query demonstrates a more complex database command that returns more than one row of results. The `query2 += ""` format builds the query statement from multiple parts.

```
query2 = "select customer_number, last_name, "
query2 += "salutation, city, year"
query2 += " from table"
query2 += " order by last_name"
```

The agent makes a connection to the database and returns the `results2` object that holds the response from the database. The `while` command loops through all the rows in the response.

```
while results2.next():
print " ",r2.getString,r2.getString,r2.getString, \
    r2.getString(4),r2.getString(5)
```

The third example shows how the agent inserts new rows into the Customers table. The setup for the database command is the same as the previous two queries.

```
insert1 = "insert into customers files (id, name, "
insert1 += "manager, phone) "
insert1 += "VALUES (1001, 'Simpson', 'Mr.', "
insert1 += "'Springfield', 2001)"
```

To make the query, the agent builds a new connection object and uses the `executeQuery` command to process the `Insert` command.

```
try:
i1 = myConn.createStatement();
ir1 = s2.executeQuery( insert1 );
except Exception, ex:
print "Exception while inserting a record: ",ex
```

The TestMaker script language provides access to all of JDBC's many methods. Consult your driver documentation for examples of additional JDBC functions.

Using Lingo to Make Test Content Close to Real

In the discussion system example presented earlier in this chapter, the `Setup_agent` creates a new discussion group by posting messages and `reply` messages. Experience shows us that when the message content is uniform, the underlying systems (Web-enabled application servers, database servers, network routers, and load balancers) try to use optimization technology to enhance the performance of the system. Rarely, if ever, in the real world would we find a discussion where every message contained the same content. Therefore, TestMaker includes a handy `Lingo` utility to automatically generate pseudo-English-like content for message subject lines and message bodies.

For example, suppose the test agent needed to create a new message. The `Lingo` utility object creates this paragraph:

```
Lipsem it quantos surbatton ditchek surbatton deplorem
ventes so surbatton quantos infreteres ipsem aqua sit
ventes. Campus ad surbatton ad infreteres ad novus surbatton
deplorem it delorum novus. Campus quantos ditchek quantos.
Via surbatton novus surbatton. Novato quantos ditchek deplo-
rem novus ditchek ventes.
```

The important thing is that Lingo creates a different paragraph each time it is called, which defeats the system's attempt to optimize performance while running tests where the message content is always the same.

Following is a test agent in its entirety that demonstrates Lingo's many uses. An explanation of the Lingo functions follows the agent.

```
# Lingo_meister.a
# Author: fcohen@pushtotest.com

# Import tells TestMaker where to find TOOL objects
from com.pushtotest.tool.util import Lingo


print
print "====================================================="
print "Technique 1: A simple message example"

myLingo = Lingo()

print "Subject:",myLingo.getSubject( 4 )
print "Message:"
print myLingo.getMessage()
print "-end of message-"

print
print "====================================================="
print "Technique 2: URL encoded message example"
print
print "If we needed to include a message in an HTTP POST"
print "command the message would need to be URL encoded."
print "Here is a Lingo message that is URL encoded:"
print
```

```
print myLingo.getMessageEncoded( 100 )

print
print "==========================================================="
print "Technique 3: A complete Lingo letter"
print
print "Here is an example business memo using several"
print "Lingo methods."
print

print "Dear Mr.",myLingo.getSingleCap(),":"
print
print "We have found the manuscript you were searching"
print "for and it includes the following excerpt:"
print

# Next we use a special version of getMessage
# to include a starting capital letter, ending period
# and encoding. The format is:
# getMessage(int size, boolean startCap,
# boolean endPeriod, boolean encoded)

print myLingo.getMessage(15, 1, 1, 0)

print
print "As you can see we are not certain what dialect the"
print "text is written in. Please do your best to let us"
print "know what it means."
print
print "Sincerely,"
print
print myLingo.getSingleCap()


print
print "==========================================================="
print "Technique 4: Making your own Lingo language"
print
print "Here is the same example business memo from"
print "technique 3 but using French words to form the"
print "Lingo gibberish."
print

caps = [ "Francois", "Sanjournais", "Berjerack", "Mon", \
"La", "Le", "Incroyable! " ]
tokens = [ "devrait","ouvrir","le","cortËge,","entourÈ", \
"d'une","bonne","partie","de","ses","adjoints","et","des",\
"dÈputÈs","parisiens","de","gauche.","les","Ècologistes", \
"seront","fortement","reprÈsentÈs","la","droite","elle,\
","ne","dÈpÍchera","que","les","quelques","membres",\
"d'on","est" ]

frenchLingo = Lingo( caps, tokens )

print "Dear Mr.",frenchLingo.getSingleCap(),":"
print
print "We have found the manuscript you were searching"
print "for and it includes the following excerpt:"
print

# Next we use a special version of getMessage which
# has options to include a starting capital letter,
# ending period and encoding. The format is:
# getMessage(int size, boolean startCap,
# boolean endPeriod, boolean encoded)

print frenchLingo.getMessage(15, 1, 1, 0)

print
print "As you can see we are not certain what dialect the"
print "text is written in. Please do your best to let us"
print "know what it means."
print
print "Sincerely,"
print
print frenchLingo.getSingleCap()

print
print "Agent done."
```

`Lingo_meister` is a simple agent to show the capabilities of the `Lingo` object provided in TestMaker's TOOL. Lingo provides English-like gibberish that is well suited to be the content of a posted test message. `Lingo` objects are created by using an `Import` command to identify the `Lingo` object to the Jython script language.

```
from com.pushtotest.tool.util import Lingo
```

The `Lingo` object is instantiated and then used by accessing its many methods.

```
myLingo = Lingo()
```

The `getSubject()` function returns a string of gibberish words with a carriage return character terminating the string. The optional value in `getSubject()` identifies how many gibberish words to include in the result.

```
print "Subject:",myLingo.getSubject( 4 )
```

The `getMessage()` function returns a random amount of paragraphs of gibberish words. Each paragraph is terminated with two carriage return characters.

```
print myLingo.getMessage()
```

When working in HTTP/HTML environments, a request to a host must often be URL encoded. In URL encoding, characters such as < and > would invalidate the HTML code so they are encoded into `&lt;` and `&gt;` codes that the host will decode. The `getMessageEncoded()` function returns a URL-encoded group of paragraphs of gibberish.

```
print myLingo.getMessageEncoded( 100 )
```

The third example in `Lingo_meister` shows how a Lingo object may be used within print commands to generate a sample memorandum letter.

```
print "Dear Mr.",myLingo.getSingleCap(),":"
```

The `getSingleCap()` method returns a single gibberish word that is capitalized. The body of the memo uses a special version of the `getMessage()` function that features options to include a starting capital letter, ending period, and encoding. The format for the special method is:

```
getMessage(int size, boolean startCap, boolean endPeriod,\
boolean encoded)
```

The last technique in `Lingo_meister` shows how an agent can define its own dictionary of gibberish words. Lingo requires a list of capitalized words and lowercase words. These are passed to Lingo when the agent instantiates a new `Lingo` object.

```
frenchLingo = Lingo( caps, tokens )
```

Now, Lingo chooses gibberish words from the new list of capital and lowercase words.

Summary

This chapter discussed system state, including the repeatable lifecycle that defines the preparation, setup, execution, and analysis stages of a test. This chapter showed TestMaker test agent script examples that use the resources of a database system to initialize the state of a system prior to testing.

The next chapter shows how to use intelligent test agents in Microsoft .NET environments and in secure environments.

9
Integrating with .NET Web Services

Microsoft delivers a broad and comprehensive solution for enterprises needing to build, deploy, and run Internet-based Web services. Microsoft's support of many popular open Internet protocols even lets them offer technology to interoperate with non-Windows systems. Microsoft built its enterprise solution on Windows NT and XP technologies over time. Before the arrival of .NET, the Microsoft acronym for all its enterprise solutions technologies was DNA, which stands for Distributed iNternet Architecture. So what is .NET?

.NET is a product strategy that overhauls and elevates the Microsoft development environment and tools to a new level of sophistication, ease-of-use, and interoperability over previous Microsoft development tools. .NET is Microsoft's way to add a new Common Language Runtime (CLR) to its object-oriented development tools and to add XML technology to everything Microsoft does.

Microsoft is working with the European Computer Manufacturers Association to put forward open standards for CLR (TG3) and the C# language (TG2). The ECMA Web site hosts working documents on these efforts: http://www.ecma-international.org/. Additional information is at http://www.dotnetexperts.com/ecma/index.html and at http://msdn.microsoft.com/net/ecma/.

.NET defines the Microsoft Intermediate Language (IL) code and CLR is the IL runtime virtual machine environment that provides memory management, automatic garbage collection, security, and threading. CLR enables

Microsoft development tools to support a variety of object-oriented languages, including C, C++, VB.NET, C#, and J#. The single CLR run-time can run components built in any of these languages that can output IL. So the Microsoft development tools work with all those languages. Following is an example C# and C++ program written in Microsoft Visual Studio .NET that features a mixture of programming languages. Visual Studio .NET compiles this program into a single unit that CLR runs.[1]

```
# We begin with a method defined using C# . . .
class Vector
{
   /// <summary>
   /// Compute the length of the vector
   /// </summary>
   public double Length
   {
      get
      {
         return(Math.Sqrt(x * x + y * y));
      }
   }
}
# . . . then add some C++ code to the mix . . .
unsafe static int AddArrayUnsafe(int[] numbers)
{
   int result = 0;
   for (int it = 0; it < Iterations; it++)
   {
      fixed (int* pNumber = numbers)
      {
         int* pCurrent = pNumber;
         int* pLimit = pNumber + numbers.Length;
         while (pCurrent < pLimit)
         {
            result += *pCurrent;
            pCurrent++;
         }
      }
   }
return result;
}
```

1. A pretty good introduction to .NET is on the Microsoft Web site at http://www.microsoft.com/net/basics/whatis.asp.

The example code shows NET's mechanism to manage C++ pointers in the otherwise memory-managed safe environment of C#. The same mechanism works in other CLR compatible languages.

In many ways .NET is Microsoft's best effort yet to use XML. XML enables Microsoft development tools to offer standardized configuration, description, and deployment definitions. For example, while Microsoft BizTalk Server, SQL Server, and Exchange Server still use the Windows Registry, they also now use XML for configuration and management features. Microsoft product strategies build XML into all its technology by coding configuration information in XML formats stored in files and databases. XML also becomes the Microsoft common language for all its APIs. In effect, it is not unreasonable for software developers to expect any Microsoft API to be available as a SOAP-based XML interface that adheres to a WSDL description.

.NET benefits IT managers, software developers, and QA analysts:

- With the emphasis on XML, configuration and management files become more open and more standardized. Third-party system management solutions such as HP's OpenView and IBM Tivoli should more easily and rapidly support Microsoft servers.
- QA analysts will find testing .NET applications easier than prior Microsoft-built applications since the APIs to these applications self-describe valid data.

Microsoft's adoption of XML and delivery of a single run-time engine enables increased productivity and efficiency as developers, IT managers, and QA analysts deliver products built with Microsoft development tools. Of course, like all new technologies, the .NET products come with problems and issues that need to be understood and tested.

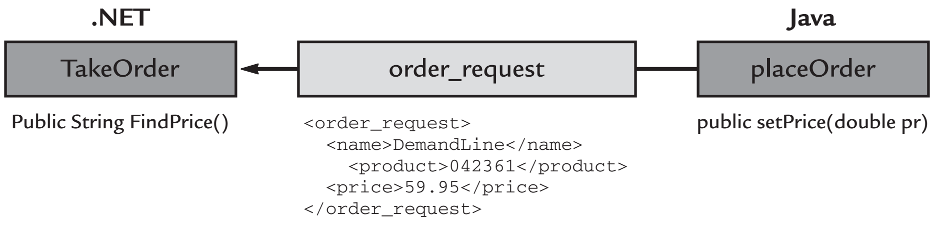
Interoperability and Integration

The good news for Java developers working with systems built with .NET is that there have been no major train wrecks. For the most part, systems built with .NET offer good interoperability with Java-based systems. The bigger problem for Java developers is usually integration.

Consider an example where a product supply company offers a Web service to take purchase orders. The service receives SOAP encoded orders.

```
<order_request>
  <name>DemandLine Purchasing, Inc.</name>
  <product>123456</product>
  <price>$59.95</price>
</order_request>
```

Figure 9–1 shows how a Java application uses the purchase service written with .NET.



```
<order_request>
  <name>DemandLine</name>
    <product>042361</product>
  <price>59.95</price>
</order_request>
```

Figure 9–1 The Java `placeOrder` function makes a SOAP request to the TakeOrder method of the published Web service.

Visual Studio .NET makes it easy to write and expose this service. The popular Java development tools available make it easy to write the `placeOrder` method to make the Web service call. Consider that this example has three integration problems:

1. The Java client function uses a floating-point double variable to hold the price while the .NET function uses a string.
2. Neither the Java client nor the .NET service explicitly know the currency expressed in the price element. For example, the Java client may send a price expressed in U.S. dollars while the .NET Web service expects Euros.
3. Java uses a naming convention where method names that begin with a capital letter are usually reserved for operating system calls. On the other hand, .NET's convention is to use capital letters to define all method names. This isn't a show stopper like the other two problems, but it may raise some eyebrows between Java and .NET developers.

These are not really interoperability issues as much as they are the systems integration issues that we typically have to deal with everyday.

.NET's support of XML and SOAP offers a solution to problems 1 and 2. By using XML Schemas, the .NET Web service provides data typing for the request method. The XML Schema below defines the price element to be a String type.

```
<xs:schema xmlns="" xmlns:xs="http://www.w3.org/2001/
XMLSchema">
  <xs:element name="order_request">
     <xs:complexType>
        <xs:sequence>
           <xs:element name="name" type="xs:string"/>
           <xs:element name="product" type="xs:int"/>
           <xs:element name="price" type="xs:string"/>
        </xs:sequence>
     </xs:complexType>
  </xs:element>
</xs:schema>
```

.NET languages use a common set of tools to publish Web services so you can expect to find a WSDL document that accompanies the service. In this example, the Java developer needs to implement a request that conforms to the schema. This likely means modifying the client-side code to express prices as strings in the request and to decode the response back into a floating-point double value.

See Chapter 7 for additional SOAP-related problems you will likely encounter, including data type interoperability, adherence to standards, and underlying platform differences. More discussion of .NET's use of WSDL comes later in this chapter.

How Is .NET Different?

By their nature, the development tools we use enforce a certain thinking that winds up in the software applications. For example, Visual Studio .NET automatically generates C# code to access a given Web service using document-style SOAP requests by default. Developers that rely solely on automated development tools risk building interoperability problems into their systems without knowing where to look when problems happen.

Document-style SOAP turns out to be a shrewd architectural decision, as we will see later. However, from a testing perspective, if your .NET Web service uses document-style SOAP encoding (the default in ASP.NET), there may be interoperability problems with other SOAP clients who only speak RPC-encoded SOAP.

Chapter 7 shows the various styles of SOAP requests and responses. The two most popular styles are RPC and Document. Most development environments include utilities that examine method declarations in source code files that define an object. Then they automatically build the needed code to offer the object's methods as Web services. The local object makes a call to a specific method and object running on a remote server and passes parameters defined in the method declaration. For example, the following C# code is automatically deployed as a SOAP-based Web service:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace myService
{
      public class myService: System.Web.Services.WebService
      {
              public HelloService()
              {
                //CODEGEN: This call is required by
                //the ASP.NET Web Services Designer
                      InitializeComponent();
              }

              [WebMethod]
              public string greetingService
              (String strMyName)
              {
                 return "Hello, " + name +", \
                 and welcome to my service.";
              }

              [WebMethod]
              public string SayGoodBye()
              {
                      return "Goodbye!";
              }
      }
}
```

This code is exposed as a Web service that has a greetingService request interface. The SOAP call receives a string as a parameter and responds with a string in its return value.

RPC-style SOAP requests are very attractive to developers because the Web service tools that generate SOAP interfaces and WSDL descriptions do a lot of heavy lifting for the developer. For example, the following method returns a complex data type:

```
public class myComplexService{
  public Hashmap greetingService( firstname name ) {
        Hashmap ht = new Hashmap();
        ht.put( "Event", name );
        ht.put( "Keycode", keycode );
        ht.put( "PcAssemblyNumber", pcassembly );
        ht.put( "SoftwareDottedString", dottedsw );
        ht.put( "CurrentTime", new Date() );
        ht.put( "CommandKey", "7" );
        return ht;
```

In an RPC-style SOAP request, the developer may depend on a development tool to build the SOAP request code to handle marshaling a Hashmap object into XML that is sent in a SOAP request to the server. This code would also make use of custom serializers that would convert objects like the Hashmap into XML.

Document-style SOAP requests begin with the XML document already defined. Instead of depending on the development tool, document-style SOAP requests are built by populating an XML tree in the application code itself.

```
public class myComplexService{
  public void callGreetingService( firstname name ) {
        body = new SOAPBody();
        body.addElement ( "Event", name );
        body.addElement ( "Keycode", keycode );
        body.addElement ( "PcAssemblyNumber", pcassembly );
        body.addElement ( "SoftwareDottedString",dottedsw);
        body.addElement ( "CurrentTime", new Date() );
        body.addElement ( "CommandKey", "" );
        . . .
        SOAPRequest.send( body );
```

.NET is a document-style environment. By default .NET development tools create document-style SOAP requests. While .NET tools provide a switch to use SOAP RPC calls, there is very little support in .NET for RPC. If an application makes a SOAP RPC call to a .NET Web service that expects a document-style request, then .NET returns a fault—the Web service equivalent of throwing an exception.

Microsoft's choice for document-style SOAP puts a bend in the direction most developers were headed. Many Web services developers come from the component software worlds that are dominated by CORBA, RMI, and COM. SOAP RPC is a natural bridge from component software architectures to Web services. On the other hand, business-to-business (B2B) exchange infrastructure has been XML based from the start so document-style SOAP is a natural there.

To see a vivid example of the divergence between Web service developers in the J2EE and .NET camps, take a look at the list of publicly available Web services published on XMethods.net. This list cleaves in two: SOAP RPC Web services are written in Java and document-style Web services are written in .NET.

One last thing to consider in both RPC and document-style SOAP calls is that the SOAP standard does not cover more complicated data elements than arrays and structures. In the previous example we would find that Microsoft has its own encoding for Hashmaps that do not interoperate with the other Web service platforms. The developer is left to encode arrays of arrays as well.

We have seen the stylistic differences between SOAP RPC and document-style SOAP. Under the covers and in the implementations of SOAP lie scalability and performance issues that come with decisions to use RPC and document-style SOAP.

Document-Style Scalability

Regardless of the SOAP call style, each SOAP message goes through the same stages of creation. The creation process begins with a value stored in an object. When the value is passed to a Web service, the object is translated into a Document Object Model element and stored in memory. When the request is marshaled to be sent to the Web service host, the Elements are rounded up and serialized into strings in XML format. The final step then gathers the XML formatted strings and sends the combined group over a transport to the Web service host. Figure 9–2 illustrates this process.

The steps depicted in Figure 9–2 show four possible places where scalability and performance problems can be introduced. The Object to Element transformation might run into problems inherent in the way that CLR repre-

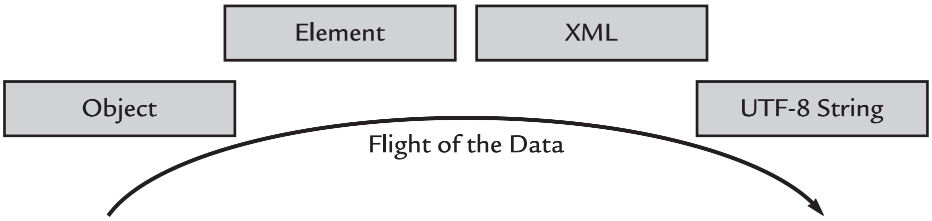


Figure 9–2 SOAP requests start with data in an object, converts the object into a DOM Element, and serializes the Element into an XML string. Then, they combine the strings and send them across a transport to the Web service host.

sents data. The Object to Element transition could introduce problems since the performance of many XML parsers is not sufficient. The serializers for the XML-to-UTF-8 transition can be susceptible to bugs and performance issues. Many other problems lower system scalability and performance.

In tests of Web service application servers, the early systems turned out to have significant scalability and performance problems, especially when comparing SOAP RPC to document-style SOAP. The results of these tests conclude the following:

- Document-style SOAP operations outperform SOAP RPC.
- SOAP RPC displays significant scalability problems as the payload size of a request and response increases.

Compare the TPS results from one popular application server:

Payload size in bytes	**RPC**	**Document**
600	168	410
3000	145	399
9000	100	404
2400	49	411
57000	18	401
96000	10	406

The test measured the round-trip time it took to make a request and receive a response as measured at the client side. To run this test in your environment, this test code and many others are available for free download at http://www.pushtotest.com/ptt/kits/index.html.

Note that in every case the TPS values for SOAP RPC transactions were slower than document-style calls. Additionally, while the document-style TPS values remained steady even under the weight of increasing payload sizes, the SOAP RPC TPS values declined significantly.

These tests do not conclude that architecturally document-style SOAP is any better than SOAP RPC, or vice versa. However, they do show that the SOAP implementations tested have serious limitations and flaws. As new and better implementations become available, a scalability and performance test of the new SOAP stacks is needed to understand how the performance characteristics change.

SOAP Headers in .NET

In the formative time of Web services, Microsoft was the first to strongly support XML Schema, WSDL, and SOAP headers. Other Web service platforms were less convinced. So to a developer on WebSphere, Apache SOAP, and BEA WebLogic, the XML Schema in WSDL documents and the contents of SOAP headers in Microsoft-built Web services may appear odd. SOAP was originally introduced in Chapter 4 with an explanation of common SOAP values and formats. This section shows how Microsoft puts its own touch on SOAP headers.

Before we explore how .NET uses the SOAP header, you may want to review the specifics of the SOAP header element. The SOAP 1.1 Specification (found at http://www.w3.org/2000/xp/Group/) and the SOAP 1.2 Working Draft indicate that a SOAP message has three possible elements: the top-level Envelope element and two child elements (Header and Body). Following is an example of a SOAP message with all three elements:

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Header>
    <PriorityHeader
      xmlns="http://msdn.microsoft.com/AYS/6/2002/">
      <Priority>HighPriority</Priority>
    </PriorityHeader>
  </soap:Header>
  <soap:Body>
    <PlaceOrder
      xmlns="http://msdn.microsoft.com/AYS/6/2002/">
      <quantity>100</quantity>
    </PlaceOrder>
  </soap:Body>
</soap:Envelope>
```

The Body element holds the main data of the request and the Header holds any metadata about the Body element. The Header is optional but it is well used in .NET applications. As time progresses, new standards for Web services will make significant use of the Header, including upcoming standards for security, credentials, routing, and referrals. .NET just happens to be a little ahead of the Header usage curve.

The SOAP specification defines three optional attributes that can apply to header blocks: the `encodingStyle` attribute, the `actor` attribute, and the `mustUnderstand` attribute. While `encodingStyle` is meant to be optional, .NET servers and the SunONE Application Server expect the `encoding-Style` attribute to be present.

SOAP is going through the same evolutionary transformation as servlets and other middleware have. While a single standalone object would respond to Web service requests, the new architecture implements several smaller objects that each handle a single facet of a Web service request, and are strung together in a string of handlers. The SOAP header elements are processed by individual .NET programs as the SOAP message moves through a chain of handlers.

The SOAP message is already defined to be in well-formed XML format, so unlike the headers in an HTTP request, the SOAP headers do not need to describe the data in the SOAP message. SOAP headers, especially in .NET, help process the data in the body. For example, in a banking environment, a SOAP header transmits authentication and transaction information that identifies the person who sent the SOAP message and the context in which it will be processed. Another example is an airline reservation service where the SOAP header defines a time limit on the validity of airfares sent in the SOAP message.

While the SOAP specification provides for any valid XML data to be included as a SOAP header, Microsoft really levers SOAP header values. So it would be safe to assume that in .NET Web services SOAP headers will play a role at improving or eroding scalability and performance.

Many times the SOAP headers in an application are particular to that single application. However, in .NET there are cases where headers will need to be processed for all Web services. For example, consider a sign-in service that is common to all Web service requests handled by a single server. For this reason .NET provides a common library called Soap Extensions. The objects you register get called before and after the serialization of a SOAP message going in and out. While this is a convenient method for .NET developers to register handlers, it also tends to introduce performance bottlenecks and scalability problems. In other platforms, you need to look at serializers for performance problems; in addition, in .NET you also need to look at the Soap Extensions that are handling Web service requests and responses.

A side effect of the .NET Soap Extensions mechanism occurs when .NET uses WSDL to define the SOAP headers. .NET developers do not likely want to make the Soap Extensions deployment information available through the published WSDL for security reasons. So, some .NET developers will omit the header definition from the WSDL and manually add the header to their requests. Others turn off WSDL generation and manually edit a static WSDL file to include the header information. These techniques are a good place to check configuration problems and should be tested and monitored routinely.

SOAP headers could turn out to be a nightmare for testing Web services. In the worst case, SOAP headers could be the dumping grounds for anything and everything related to a Web service. Building intelligent test agents that understand and validate SOAP headers is an important part of testing and monitoring Web services.

WSDL .NET Style

Chapter 7 described WSDL as an XML-based document format that describes the services a Web service provides so tools can access and use them. The WSDL for a service describes the location of the service; the names of the available services; their method names, parameters, and types; and return values. Microsoft's ringing endorsement of WSDL as the standard comes through loud and strong in the .NET development tools—they all use WSDL to describe a SOAP interface to XML data.

Microsoft has two additional efforts underway to describe SOAP interfaces: the Web Services Meta Language (WSML) and the Web Services Discovery Tool (Disco). WSML is another XML-based document format used by some Web service tools—Microsoft SOAP Toolkit for one—which provides mappings between the Web service methods and their parameters and the underlying code object, such as a COM component, that provides the service.

Disco is an XML document format that contains entries pointing to WSDL resources describing available Web services. Disco gives IT managers a way to centralize the listing of multiple Web service providers. With Disco everything goes into a single document that tools can use to select the desired Web service and obtain a reference to it.

As testers we need to recognize when platform vendors try to stretch, strain, or even bully a standard through their implementation. WSML and Disco are examples of this kind of behavior at work. Even if their effort does not become part of the standard, Microsoft extensions to the standard will become part of your test plan for .NET-based Web services. Consider how Microsoft made its mark in the style it uses to write WSDL descriptions of .NET services.

To illustrate the differences found in a .NET WSDL description of a Web service, consider a real-world live hosted .NET Web service from Service Objects Inc., found on the Web at http://www.serviceobjects.com. Service Objects provides several services used by Web sites and portals to provide personalization and content of interest to a site's readers. The example below focuses on the GeoCash service. GeoCash takes a U.S. postal zip code and returns a list of automatic teller machines (ATMs) near the zip code. Service Objects sells this service to people with Web sites, corporate portals, and application vendors.

For the WSDL full definition for the GeoCash service, go to http://ws.serviceobjects.net/gc/GeoCash.asmx?WSDL. The first time I looked at a WSDL document my eyes went buggy, my brain filled with confusion, and I found myself trying to make sense of it all. It took about 6 months to be able to look through WSDL and see a Web service. Let us look at various pieces of this WSDL document to find the GeoCash service.

Our first clear indication that this is a .NET Web service comes in the last section.

```
<binding name="GeoCashSoap" type="s0:GeoCashSoap">
  <soap:binding transport =
 "http://schemas.xmlsoap.org/soap/http"
 style="document" />
  <operation name="GetATMLocations">
    <soap:operation soapAction =
    "http://www.serviceobjects.com/GetATMLocations"
    style="document" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
</binding>
```

Here we see that the `GeoCashSoap` binding defines a document-style SOAP request and response that use literal encoding. Literal encoding tells the SOAP stack to take the body of the SOAP request and response as an already formatted DOM tree and serialize it all at once into XML.

Another indication that .NET created this WSDL document is the <types> branch.

```
<types>
  <s:schema elementFormDefault="qualified"
  targetNamespace="http://www.serviceobjects.com/">
    <s:element name="GetATMLocations">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1"
          name="strInput" type="s:string" />
          <s:element minOccurs="0" maxOccurs="1"
          name="strLicenseKey" type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
```

.NET uses XML Schema to define type information for the parameters of a request and response. The types element shows the valid parameters allowed to make requests of the service. In this example, the request takes two String values for the zip code and a License Key. We get the License Key from Service Objects directly.

The next group of type definitions defines a hierarchy of complex data types that comes in the response document. The response document contains a single `GetATMLocationsResult` element . . .

```
<s:element name="GetATMLocationsResponse">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="0" maxOccurs="1"
      name="GetATMLocationsResult"
      type="s0:ATMLocations" />
    </s:sequence>
  </s:complexType>
</s:element>
```

that has a single `ATMLocations` element . . .

```
<s:complexType name="ATMLocations">
  <s:sequence>
    <s:element minOccurs="0" maxOccurs="unbounded"
    name="ATM" type="s0:ATM" />
    <s:element minOccurs="0" maxOccurs="1"
    name="Error" type="s:string" />
  </s:sequence>
</s:complexType>
```

that contains a series of ATM elements. Each ATM element contains information about a single bank, including the address, location, and notes about the ATM machine.

One very handy feature of .NET-created Web services is a friendly HTML front end to describe and access a Web service. Even with the WSDL description to the GeoCash service, many times it is handy to have an HTML description of the service. For an example of the GeoCash HTML description, go to:

```
http://ws.serviceobjects.net/gc/GeoCash.asmx?op=GetATMLocations
```

A Test Agent for .NET Environments

Next we will see a TestMaker test agent that makes a document-style .NET Web service request to the GeoCash service. Many of the values in this agent were found in the WSDL document for the GeoCash service that was presented in the previous section of this chapter. Following is the agent in its entirety, and an explanation of how it works.

```
# GeoCash_agent.a
# Author: fcohen@pushtotest.com

# Import tells TestMaker where to find Tool objects
from com.pushtotest.tool.protocolhandler import \
ProtocolHandler, SOAPHeader, SOAPBody, SOAPProtocol
from com.pushtotest.tool.response import Response

# This agent also uses JDOM APIs to handle XML data
from org.jdom import Document, Element, JDOMException,\
Namespace, DocType
from org.jdom.output import DOMOutputter, XMLOutputter
from org.jdom.input import SAXBuilder
from java.io import StringReader


# Variables definitions

zipcode = "95008"

license = "0"

host = "ws.serviceobjects.net"
port = 80
path = "gc/GeoCash.asmx"


# Helper function to build a request XML document

def addElement( top, name, content, ns ):
    newelement = Element( name, ns )
    newelement.addContent( content )
    top.addContent( newelement )


# Main body of agent

print "Processing..."
print
```

```
# Set-up a SOAPProtocol object and populate it with the
# info to make a request to the GeoCash service

soap = ProtocolHandler.getProtocol("soap")

soap.setHost( host )
soap.setPath( path )
soap.setPort( port )


body = ProtocolHandler.getBody("soap")

body.setMethod("GetATMLocations")

soap.setActionURI( "http://www.serviceobjects.com/GetATMLo-
cations" )

# Build the request XML document

xmlns1 = "http://www.serviceobjects.com/"

# Create the request document by first creating
# the Respond element ...
elOne = Element( "GetATMLocations", xmlns1 )

# ... add the input elements ...
addElement( elOne, "strInput", zipcode, xmlns1 )
addElement( elOne, "strLicenseKey", license, xmlns1 )

# ... then tell the SOAPProtocol handler about the request
document.
doc = Document( elOne )
body.setDocument( doc )

# Finally, attach the body object to the soap protocol han-
dler object
soap.setBody(body)

print "Here is the XML request document:"
xo = XMLOutputter()
print xo.outputString( doc )
print

# Send the request to the .NET host and receive the response

response = soap.connect()
```

```
# Parse through the responses

print "Here is the raw response:"
print response
print

builder = SAXBuilder()
doc = builder.build( StringReader( response.toString() ) )

print "Here are the elements in the response:"
print

envelopeElement = doc.getRootElement()

# Document-style SOAP responses will use the
# document-literal type of namespace for the Body element
soap_ns = Namespace.getNamespace \
("soap", "http://schemas.xmlsoap.org/soap/envelope/")

bodyElement = envelopeElement.getChild("Body", soap_ns)

# The entries in the body use the Service Objects namespace
# which also tells us which .NET service responded
so_ns = Namespace.getNamespace( "", "http://www.serviceob-
jects.com/")

atmlocrespElement = bodyElement.getChild( \
"GetATMLocationsResponse", so_ns )
print "atmlocrespElement=",atmlocrespElement

atmlocresultElement = atmlocrespElement.getChild( \
"GetATMLocationsResult", so_ns )
print "atmlocresultElement=",atmlocresultElement

# Notice that GetATM
atmlist = atmlocresultElement.getChildren( "ATM", so_ns )
print "atmlist=",atmlist
print

for i in atmlist:
    print "  ",i.getName()
    print "  ",i.getChildText("Bank", so_ns)
    print "  ",i.getChildText("Address", so_ns)
    print "  ",i.getChildText("Location", so_ns)
    print "  ",i.getChildText("City", so_ns), \
    i.getChildText("State", so_ns), i.getChildText("Zip",\
    so_ns)
    print "   Longitude:",i.getChildText("Longitude", \
    so_ns)," Latitude:",i.getChildText("Latitude", so_ns)
    print "  ",i.getChildText("Notes", so_ns)
    print


# All done

print
print "Agent ended."
```

TestMaker runs this agent. The agent connects to the Service Objects host and requests the ATM locations for Campbell, California. The response should contain 10 ATM machines in various locations in Campbell. Let's look at the following to see how this agent works:

```
from com.pushtotest.tool.protocolhandler import \
ProtocolHandler, SOAPHeader, SOAPBody, SOAPProtocol
from com.pushtotest.tool.response import Response
```

First we import the `SOAPProtocol` handling objects from the TestMaker TOOL.

```
from org.jdom import Document, Element, JDOMException,\
Namespace, DocType
from org.jdom.output import DOMOutputter, XMLOutputter
from org.jdom.input import SAXBuilder
from java.io import StringReader
```

We also import the JDOM objects to construct and parse the XML data. JDOM is a simple interface to manipulate XML data.

```
zipcode = "95008"
license = "0"
```

These variables define the U.S. postal zip code and license number to send to the Web service host. Service Objects provides licenses to control services to paying customers. In support of this book, Service Objects was nice enough to provide a special license number for the test agent presented in this chapter. Please do not abuse this privilege by reusing the license number or create excessive requests to the Service Objects host.

```
host = "ws.serviceobjects.net"
port = 80
path = "gc/GeoCash.asmx"
```

These variables define the destination server and location of the Web service. We learned these values from the WSDL definition in the previous section of this chapter.

```
def addElement( top, name, content, ns ):
    newelement = Element( name, ns )
    newelement.addContent( content )
    top.addContent( newelement )
```

`addElement` is a helper function that takes the DOM element and appends a new child element. This is just a convenience function to make the subsequent `addElement()` commands more legible.

```
soap = ProtocolHandler.getProtocol("soap")
body = ProtocolHandler.getBody("soap")
```

The TOOL in TestMaker provides a `SOAPProtocol` handler object and SOAPBody object. We use these to construct the request document and to communicate with the host.

```
body.setMethod("GetATMLocations")
soap.setActionURI( \
"http://www.serviceobjects.com/GetATMLocations" )
```

The WSDL document tells us to use the `GetATMLocations` method to request a list of 10 ATM locations for a given zip code.

```
xmlns1 = http://www.serviceobjects.com/
```

This references the Namespace of the parameters in the request. In document-style SOAP requests, TestMaker uses the Namespace of the first element <Respond> to determine the destination service name. The endpoint gets the request to the right server. The Namespace of the first element gets us to the right Web service.

The WSDL told us that GeoCash wants a zip code and license element. The agent uses the JDOM APIs to build the XML request in a DOM object.

```
elOne = Element( "GetATMLocations", xmlns1 )
addElement( elOne, "strInput", zipcode, xmlns1 )
addElement( elOne, "strLicenseKey", license, xmlns1 )
```

Here the agent adds the two request elements using the addElement convenience method.

```
doc = Document( elOne )
body.setDocument( doc )
soap.setBody(body)
```

The agent then passes the DOM object doc to the SOAPBody object to make the SOAP call to the server.

```
response = soap.connect()
```

"Houston, this agent is ready for lift-off." The connect() method sends the request to the Service Objects host and returns a response object. The response object is in XML format. The agent uses JDOM APIs to look into the response document to find and validate the ATM location data.

```
builder = SAXBuilder()
doc = builder.build( StringReader( response.toString() ) )
```

Use the JDOM SAXBuilder object to create a new DOM object that parses through the XML data we received from the .NET host. Recall that the WSDL (http://ws.serviceobjects.net/gc/GeoCash.asmx?WSDL") for the response document tells us to expect the response XML to look like this:

```
Envelope
   Body
     GetATMLocationsResponse
       GetAMTLocationsResult
         ATM
         ATM
         ...
```

Look at how the agent uses JDOM APIs to parse through the response document contents.

```
envelopeElement = doc.getRootElement()
```

First, we find the root element. By convention every SOAP response document names the root element to Envelope.

```
soap_ns = Namespace.getNamespace("soap", \
"http://schemas.xmlsoap.org/soap/envelope/")
```

The agent creates a `Namespace` object that is used to retrieve the Body element. The Body of the SOAP response is encoded in document-literal style as defined by the XML schema definition at http://schemas.xmlsoap.org/soap/envelope/.

```
bodyElement = envelopeElement.getChild("Body", soap_ns)
```

Here we get the Body element using the document-literal style namespace.

```
so_ns = Namespace.getNamespace( "", \
"http://www.serviceobjects.com/")
```

The agent creates a second `Namespace` object to retrieve response elements emitted by the Service Objects host.

```
atmlocrespElement = bodyElement.getChild( "GetATMLocations-
Response", so_ns )
atmlocresultElement = atmlocrespElement.getChild( \
"GetATMLocationsResult", so_ns )
```

The agent jumps down two nested children elements for `GetATMLocationsResponse` and `GetATMLocationsResult`.

```
atmlist = atmlocresultElement.getChildren( "ATM", so_ns )
```

We are now down to the `GetATMLocationsResult` element whose children elements are a list of ATM elements; one ATM element holds the information on one teller machine.

```
for i in atmlist:
    print "  ",i.getName()
    print "  ",i.getChildText("Bank", so_ns)
```

The agent uses a convenient `for/in` statement to loop through all the children elements. Each time through the loop the `i` variable points to the next ATM element.

This agent is just a start, a skeleton, and paper-thin. An expanded agent makes multiple requests of various sizes and delays, operates multiple concurrent threads of the request script to increase the load on the server, runs multiple copies of the multithreaded agent on multiple machines, and adds simple logging functions to the agents so we may determine TPS and SPI values. Chapters 11 and 12 provide detailed examples of building a complete testing system using intelligent test agents.

Near Term Considerations

For a long time to come, .NET will be the new kid on the block. As such, there are a few near term considerations to keep in mind when designing and testing Web-enabled applications that work with .NET.

First, consider that solving interoperability problems is different from platform to platform. When a developer faces interoperability problems in Java platforms, the underlying code is usually available to find and solve the problem. Microsoft development tools, applications, and operating systems are closed to such public scrutiny.

Considering the closed-source origins of Microsoft, I find it impressive to see Microsoft taking its first steps toward an open code review. Microsoft now offers a "shared source" concept where the source code to their tools is available under a special license. Details on this are found at http://www.microsoft.com/licensing/sharedsource/default.asp and http://msdn.microsoft.com/library/default.asp?url=/library/en-us/Dndotnet/html/mssharsourcecli.asp. Some Microsoft libraries even offer a public view of their source code. For example, the native C++ ATL implementation of SOAP—which does not run on the CLR—is available on the Microsoft Web site.

Interoperability between computer systems, software applications, and operating systems is more art than science. Interoperability gets better the more it is practiced. So, it follows that .NET interoperability will improve over time, provided Microsoft continues to improve .NET. Microsoft regularly participates in SOAPBuilders, a loose affiliation of Web service vendors whose goal is to proof interoperability. SOAPBuilders also publishes an interoperability test suite for checking SOAP implementations. You can access this suite at http://www.xmethods.com/ilab/ and http://www.whitemesa.com/interop.htm.

Lastly, consider that the Windows development community is undergoing a significant change. Visual Basic developers are moving from a script-oriented language to an object-oriented programming language like C#. Learning object programming is usually a difficult and long task and will have an impact on testing Web-enabled applications. As an example, I participated in one of Microsoft's early day-long conferences to show .NET development tools to a group of Visual Basic developers. One presentation showed how .NET development tools allow programs written in C# to include the script-oriented Visual Basic commands to be inserted into a block of code wrapped with *unsafe* tags. One VB developer remarked to me afterward that the only thing about C# that he understood was the Visual Basic code between the *unsafe* tags.

Microsoft is betting that it can convince script-oriented developers to learn object-oriented programming by showing how .NET development tools will make them more productive. Visual Basic developers who make the jump to VB.NET will benefit from the world of Web services now available to their code. For example, instead of writing a security manager, the VB.NET developer can use Passport.

VB.NET software written by newbie object-oriented developers often includes the following problems that may be detected in a test:

- **Functions that spawn many threads.** Script languages usually handle threading for the developer. So thread deadlocks, concurrency problems, and race conditions are easily accomplished with object-oriented programming.
- **Method declarations with complex data types.** Method declarations in object-oriented programs have the advantage of using complex data types—for example, a customer record containing numbers, dates, and purchase history—that may cause overly complicated SOAP requests.
- **Exception handling in boundary conditions.** Object-oriented programs follow a stack of calls from one object to another. Unwinding a stack when an exception is thrown is often problematic for the novice.

Summary

In this chapter we learned how Microsoft's adoption of XML and delivery of a single run-time engine has elevated Microsoft development tools to new levels of flexibility, productivity, and efficiency. The chapter covered several issues specific to testing Web services built in .NET.

Over time it will be exciting to see how Microsoft's CLR engine handles production environments. Really large .NET scalability is full of unknowns today. Consider that .NET sits on top of Windows XP—itself new technology—and on top of hardware with little history of high-volume situations.

Additionally, it will be exciting to see how the open source community helps .NET to become a cross-platform platform. Efforts like Go-Mono are building an open source implementation of the .NET development framework. Details are at http://www.go-mono.com.

10 Building and Testing Intranets and Secure Environments

Previous chapters showed techniques to test Web-enabled applications using a variety of connection protocols. The techniques assumed open and free connectivity from the test machine to the host. Prior to the terrorist attacks of September 11, 2001, the computer industry was doing well delivering inexpensive routing and firewall devices that both protected network resources and provided network managers with load balancing and failover functions. Additionally, digital certificates, public key encryption, Public Key Infrastructure (PKI), and application security were generally improving and gradually being adopted. The terrorist attacks, Anthrax scares, suspicions of worldwide conspiracies, and a new global activism give security a whole new level of priority among software developers, QA technicians, IT managers, and users. Testing Web-enabled applications in secure environments requires an understanding of security infrastructures' effect on tests and techniques for testing secure Web-enabled applications.

Getting a Head Start

This chapter is not an all-encompassing treatise on security in Web-enabled applications. It cannot be so because security technology is too fast moving in today's environment. Rather, this chapter intends to give the developer, QA technician, and IT manager a head start on testing in secure environments by presenting practical tips and techniques for testing Web-enabled applications in secure environments.

In my experience security infrastructure least impacts system scalability, performance, and functionality. Few tests really push the limits of firewalls and routing devices because the tests often find other scalability and performance bottlenecks first. While conducting functional testing, most security infrastructure technology will work again and again once it works the first time. And in general, the network-level security devices (firewalls and routers) have the most bandwidth and capacity of all the components that serve the system.

Unfortunately, while security infrastructure protects us from hackers and terrorists, security infrastructure has a dark and brooding side too. While a tester's motto is "trust after verification," the security expert's motto is "trust no one." This permeates into the security software and devices themselves. Friendly graphical interfaces are anathema in the security world.

In my experience configuration and setup of secure infrastructures prior to conducting a test becomes the single most time-consuming part of the overall test. With rare exception the devices and software found in security infrastructure offers complicated, elaborate, and nonstandard configuration protocols. It is enough to make one gray and bald, which in my case came early in my career.

This chapter presents testing issues where network routing, virtual private networks, and network segmentation must be navigated to achieve meaningful test results. Then, we explore transport-level security that uses SSL and HTTP basic authentication protocols. This book provides pointers to emerging security standards, including SAML, WS-Security, and XML Signature. For additional reading materials on how TestMaker supports these higher-level security standards, see the PushToTest Web site.

Security by Routing

To understand network-layer security, I show how a network manager's choices for TCP/IP connectivity and routing have changed over the years. Figure 10–1 shows the most basic Web environment where a Web client connects over a public TCP/IP network directly to a computer running Web server software. In this configuration any device on the Internet may open a socket to the host. The Internet began with this as the dominant configuration but it didn't take long before this configuration proved problematic.

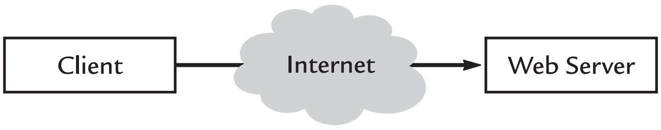


Figure 10–1 Basic network connections for open Web sites put the client in direct connection to the server.

In the mid-1990s most Internet network traffic was concentrated in the hands of a few backbone providers. When a backbone provider made a router configuration mistake, the story became headline news. Broken routing tables could misdirect all the traffic on a network to a single open server and make the Web applications hosted on the server inaccessible. Very quickly network operators and Internet service providers began using routers that offered flexible policies to control access. For example, one popular router policy prevents multiple incoming ping requests from a single source to protect from denial-of-service attacks. Today network managers implement hundreds of routing policies as part of an overall plan to secure network traffic. When beginning to test a Web-enabled application system, experience shows that a conversation with the network manager is a wise choice to avoid problems later on.

Another big change from the early days of the Internet involves using Network Address Translation (NAT) and Dynamic Host Control Protocol (DHCP). Figure 10–2 shows a network environment where the client connects to the Internet through a NAT/DHCP router. The DHCP service assigns the client machine a valid TCP/IP address. When the client makes a request using the TCP/IP address, the router translates the IP address into a publicly accessible Internet address and makes the request as a proxy to the client.

NAT was highly controversial in the early days of the Internet because it made network management more difficult. For example, if a network manager tried to ping the client from the server, the response to the ping came

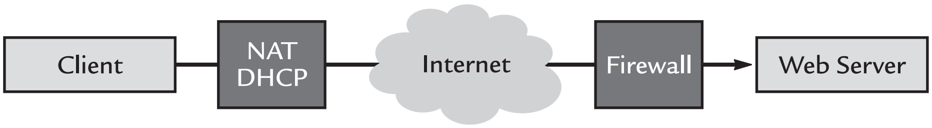


Figure 10–2 Routers that provide Network Address Translation and firewall functions also provide a form of security that protects the client machine from being hacked.

from the NAT router and not the client. Today, NAT is hugely widespread. For example, 2Wire manufacturers a popular home DSL modem/router that includes a NAT and DHCP server that is sold by the major ISPs.

Firewalls, NAT service, and router devices are oriented to solve routing problems at the network layer; however, they have a direct impact on security because they make IP address-based security very difficult. As a consequence, many efforts to implement security in Web-enabled applications has moved up to the application layer.

Virtual Private Networks

A Virtual Private Network (VPN) is a way of tunneling traffic from one network (typically a company internal network) over another network (often the Internet) so that it appears that you are connected to the company network directly. To protect the content of the data that moves through the VPN, most VPNs use digital certificates for authentication and encryption technology to protect the tunneled data.

Figure 10–3 shows an example of the communication from client to server using VPN technology.

Initially, VPNs were built by running security software on the client machine and server. Later VPN appliances—small router-like devices—began to appear that provided the VPN encryption and authentication functions but did not sap the client or server machine's bandwidth. Today, VPN software comes in many flavors and varieties, including VPN software that runs on the client machine, in a dedicated VPN device, and as part of a firewall and router.

Unfortunately, each type of VPN system usually does not have a programmatic interface for a test agent to configure the VPN for its own use. For example, I ran into problems while testing an XML-RPC single-sign-on service that connected an intelligent test agent to a server running on a private secure corporate network. In this setting an XML-RPC request originated from outside of the network and required a VPN authorization key stored in the corporate

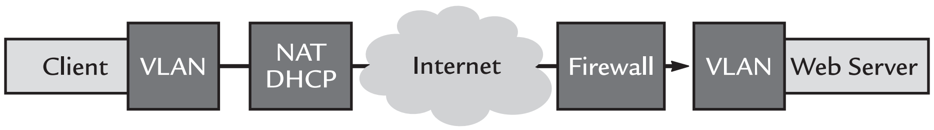


Figure 10–3 Software at the client and server devices to encrypt data communicated over a Web infrastructure forms a VPN.

database. In this example, the solution was to retrieve a small cache of key codes using a batch script that creates a delimited data file. The test agent looked into the data file to get the needed key as the test progressed.

VPN scalability and performance is often overlooked with the expectation that the application server or database server will be the largest performance and scalability bottleneck. While that may be the case, using intelligent test agents help you to learn just how much performance overhead is built into the VPN devices, software, and implementations. Experience shows us the surprising value of testing a system with and without a VPN to learn the VPN's performance overhead and scalability limitations.

Network Segments and Subnets

Network managers seeking an additional level of security have put parts of a system onto separate networks that are unreachable from one another. Figure 10–4 shows a popular server-side architecture that uses a Web server, application server, and database server to meet user demands. Each server is put onto a physically isolated network by installing multiple network interface cards (NICs) into each machine. Even if someone were to hack into the Web server, the application server is on a different physical network and would prove to be another equally difficult challenge to hack into.

Of course the network segments introduced in Figure 10–4 create problems for conducting tests. Imagine running a functionality test that results in faults at the application server stage of a request. The intelligent test agent does not have direct connectivity to the application server since they are on separate networks. One possible solution is to put intelligent test agents on each network segment to act as service monitors. In such a system the intelligent agent sits on the application server network segment. Another solution is to run the test agents on a single machine that contains multiple NICs—one network interface per network segment—for multihoming.

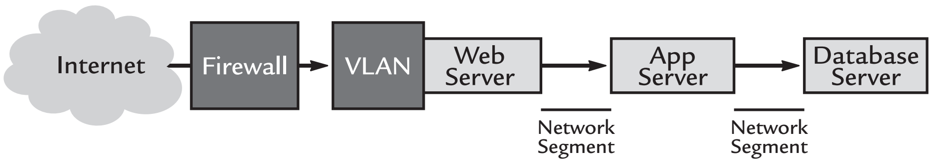


Figure 10–4 Focusing on the server side of a secure Web environment, the Web, application, and database servers are often put on separate network segments to disable direct access from outside the local area network.

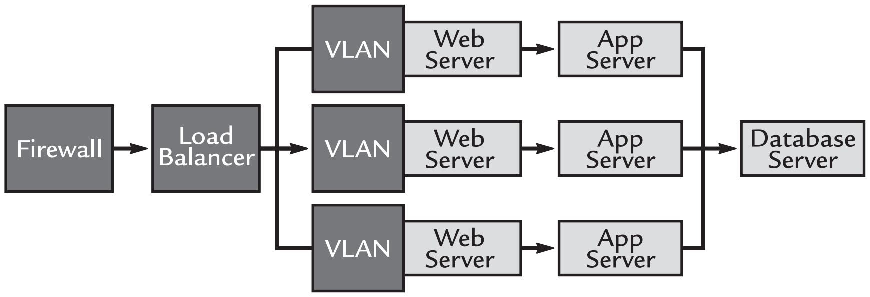


Figure 10–5 Load balancers are dynamic routers that assign and manage affinity from user machines to a host for the duration of a user session.

The following describes a network design for scalability that uses a load balancer and several identical hosts to handle incoming requests as the Flapjacks architecture. Figure 10–5 shows the Flapjacks architecture in a secure environment where VPN technology is used upstream of the load balancer.

Several load balancers also support VPN and SSL technology to make it possible to offload VPN demands on CPU bandwidth to the dedicated load balancer device. Being able to multihome is important in a Flapjacks architecture. Without multihoming, all the traffic generated from an intelligent test agent would be routed by the load balancer to a single Web host. Unfortunately, this may not provide meaningful test results.

Load balancing often deserves its own handlers. For example, I provided testing services to a customer using a load balancer to share the load among a dozen application servers for an XML-RPC-based system. Their application software was only moderately reliable. Consequently, the network manager would cycle each application server every night. The network manager would wait for all current sessions on an application server to end before cycling the server. A healthy proportion of users were making requests from behind a NAT router. The load balancer mistakenly considered all requests from the users behind the NAT router as sharing the same session. To the network manager the server appeared to never finish serving the session. Switching the load balancer to use cookie-based session management provided accurate session management even with traffic coming from a NAT router.

So far this chapter has concentrated on network- and user-level security in local and wide area networks. Next we look at transport-level security.

Transport Security

As businesses rushed to integrate Web-based applications into their information systems, they made transport layer security a requirement for Web infrastructure. The SSL protocol is the most widely used and unofficial standard for securing the information exchange between applications and end users.

The specifications for SOAP and HTTP purposefully do not describe mechanisms for securing requests and responses. Into the SOAP security vacuum, several security technologies and strategies emerge, including HTTP over SSL connections, SOAP over SSL connections, .NET Passport authentication, and HTTP basic authentication. In Web services, security needs are met with WS-Security and SAML.

HTTP over SSL Connections

SSL is the cornerstone of information security technology for most of the businesses in the world today. SSL provides these features when used in business applications:

- Authenticates that the server is who it says it is
- Keeps private any communication between applications and users

Additionally, with a little more effort, the digital certificates SSL use may also be used to prove that transmitted data has not been tampered with and prove the identity of the user to the server.

SSL uses digital keys to accomplish these goals. The keys are often based on very large prime numbers—as in the case of RSA and Verisign. SSL uses two keys: a *private key* that only you have access to and a *public key*, which can be accessed by anyone.

The private and public keys work together. For example, with RSA keys, a message encrypted with a private key can only be unscrambled with the public key. The longer the key, the more difficult it becomes for someone to hack into the key.

In the real world we prove our identity by carrying credentials—driver's license, credit cards, and passports—and have fairly unique handwritten signatures. Our own personal keys act as a signature in the digital world of SSL. To digitally sign a document, a hash is made of the document and then signed with the user's private key. To verify the signature of the document later, you run a second hash on the document and make sure it matches the original hash.

In the 1990s the computer industry supported the X.509 recommendation to define a standard for digital certificates. X.509 is actually an International Telecommunication Union (ITU) recommendation, which means that it has not yet been officially defined or approved. As a result, companies have implemented the standard in different ways. For example, both Netscape and Microsoft use X.509 certificates to implement SSL in their Web servers and browsers. But an X.509 certificate generated by Netscape might not be readable by Microsoft products, and vice versa. For details on the ITU, go to http://www.itu.int/home/index.html. Using X.509 certificates in the Test-Maker framework is described further later in this chapter.

So who verifies the keys? That is up to a digital certificate authority. A user generates a private key, and after verifying the user's identity, the certificate authority signs the user's public key with its own private key. The combination of the user's public key and the signature of the certificate authority completes the digital certificate.

Imagine that the user wants to send a confidential email. The user encrypts the message with the help of the recipient's public key, which is stored in the recipient's certificate. The recipient uses his or her own private key to decrypt the message. Since the recipient is the only one with access to the private key, only he or she can decrypt the message.

The same SSL system works in Web service environments. When using a browser to buy products online, public key digital signing technology provides security and authentication services in an HTTP/HTML environment. Figure 10–6 describes the SSL interaction between client and server.

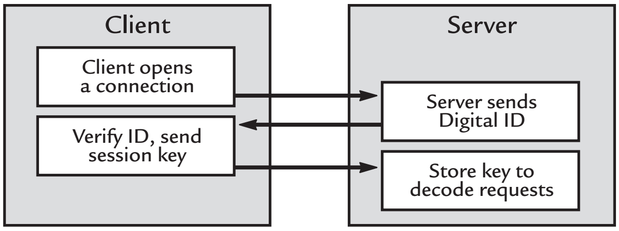

Client
Server
Client opens a connection
Server sends Digital ID
Verify ID, send session key
Store key to decode requests

Figure 10–6 Making an HTTP request over an SSL connection begins a process by which the client and server exchange digital certificates to authenticate each other. Then, they use the same certificates to encrypt the data.

In an example of SSL in an ecommerce scenario, consider a user that needs to order a product securely over an Internet connection. The browser opens a connection to the Web host and requests the server's digital identity. The server responds with a copy of the public key of its server certificate that contains its public key. The browser verifies the key and sends a session key that is valid only during the current session. The server stores the key and uses the key to decode subsequent requests.

TestMaker supports SSL authentication and encryption in its protocol handler objects for HTTP, HTTPS, and SOAP protocols. Here is an agent showing how to make a request to a Web host using the HTTPS protocol, which uses SSL encryption:

```
# HTTPS_Connect.a
# Author: fcohen@pushtotest.com

print "Agent running: HTTPS_Connect.a"
print "Description:"
print "  Connect to a Web host using an SSL connection."
print

from com.pushtotest.tool.protocolhandler import \
HTTPProtocol
from com.pushtotest.tool.response import Response

# Create an HTTP object to communicate with the host
# over an SSL encrypted connection
protocol = HTTPProtocol()
protocol.setScheme( "https" )

# Tell it where the host and document are located
protocol.setHost("citibusinessonline.da-us.citibank.com")
protocol.setPath("basprod/citiiwt/BAbuscde.html")

# Get the document
response = protocol.connect()

print "Received this response from the secure host:"
print response.toString()

print "Timing values for this request:"
print "Total: " + Long(response.getTotalTime()).toString()
print "Data : " + Long(response.getDataTime()).toString()
print "Setup: " + Long(response.getSetupTime()).toString()

print "Agent complete."
```

The HTTPS_Connect agent connects to the sign-in screen of Citibank's online portal. The agent makes a simple HTTP `GET` request for the sign-in page, displays, and response, and the time it took to perform the `GET` operation and then exits.

```
from com.pushtotest.tool.protocolhandler \
import HTTPProtocol
```

The import statement tells TestMaker that the agent will be using the basic `HTTPProtocol` object. For details on the `HTTPProtocol` object and its many functions, see Chapter 3.

```
protocol = HTTPProtocol()
protocol.setScheme( "https" )
```

The `httphandler` variable now points to a new `HTTPProtocol` object. This object handles all communication with the HTTP/HTML Web-enabled application. The `protocol.setScheme()` method tells the `HTTPProtocol` object to secure the request to the host using the HTTPS protocol that uses SSL digital certificates. TestMaker uses the Java Secure Sockets Extension (JSSE) library in Java 1.4 to provide the HTTPS protocol. Details on JSSE are at http://java.sun.com/products/jsse/index-14.html.

```
protocol.setHost("citibusinessonline.da-us.citibank.com")
protocol.setPath("basprod/citiiwt/BAbuscde.html")
```

The `setHost()` and `setPath()` methods tell the `HTTPProtocol` where to find the host and document.

```
response = protocol.connect()
```

The `connect()` method tells the `HTTPProtocol` object to make the request to the host. The `HTTPProtocol` object returns a new `response` object containing the HTTP response from the host.

Since this connection is being made over a connection secured using SSL, you may notice that it takes quite a bit longer than the non-secure HTTP request.

```
print response.toString()
```

This prints the `response` object to the TestMaker output window.

The HTTPS_Connect agent gives a very simple example of interacting with a Web host over an SSL connection. We will see later that SOAP-based Web-enabled application requests and responses may be made using SSL too. First, we investigate the problems SSL potentially adds to a Web-enabled application environment.

What Usually Goes Wrong in SSL Systems

SSL is widely implemented as a software-based solution. In this implementation, the server negotiates the SSL connections with clients and performs encryption and decryption at the software level. The SSL handshake, in which the client and the secure server negotiate the choice of algorithm and keys, is an extremely processor-intensive operation. Running the `HTTP _connect` test agent in the previous section demonstrates the overhead of a secure connection. At some point the server runs out of resources (CPU bandwidth, memory, or network bandwidth) when performing the handshake to run the secure application. Network managers often overcome this limitation by adding more powerful servers.

Another way of going is to add an individual SSL accelerator card to the server. Experience shows that while SSL traffic does not significantly affect overall network traffic, it does tax the servers that process network traffic. Adding an SSL accelerator card to each server offloads the SSL work to the coprocessor on the card and enables the Web application on the server to respond in normal time.

In a Flapjacks environment, network managers find that the extra expense of adding SSL accelerator cards to every server can become overwhelming. Each server handling encrypted content requires an SSL accelerator card and a digital certificate. To obtain a digital certificate for each server from a certificate authority (CA), administrators must create a public key–private key pair and a Certificate Signing Request (CSR), and then submit these items to a CA. This process must be repeated for every server in production. Digital certificates are valid for a limited time only, and administrators must renew each certificate each time it expires.

Not only is the management of these SSL-enabled servers time-intensive, it is also costly. Technological advances have reduced the cost of SSL accelerator cards, but they are still expensive. And each time a certificate expires, it must be repurchased from the certificate authority. These expenses significantly increase the total cost of procuring and managing the SSL-enabled servers. As a result, while testing Web-enabled applications in secure environments, the test will usually turn up one or two expired certificates. While it is possible to continue testing using temporary certificates, the results may not be meaningful.

Like any security technology, digital signatures used in the SSL model are fallible. If the certificate authority's root key is stolen, then anyone can create digital certificates, which compromises the trust level of the certificate authority and makes all the certificates from that certificate authority null and void. Certificate authorities go to great lengths to keep their keys secure, including armored bunkers. Additionally, if the user loses his private key or if it is stolen, then anyone possessing the private key can assume the user's identity.

Unfortunately, in today's business environment SSL is not universally used. But it is becoming more accepted. Languages, platforms, and certificate authorities now all support SSL and work hard at reducing all the complexity behind keys, hashes, and digital certificates.

SOAP over SSL

The SOAP specification purposefully avoids describing mechanisms for securing requests and responses. The early SOAP-based Web-enabled applications use SSL to encrypt the SOAP request and response documents. Later in this chapter we will see why SSL is not very well suited to secure SOAP-based Web-enabled applications. But for now we show how to use SSL in a SOAP environment.

SOAP implementations run on top of application server and Web server technology. It should be no surprise then that the SOAP requests may be made over SSL-encrypted connections. The TestMaker framework builds intelligent test agents that can communicate with Web-enabled application hosts using SOAP protocols over SSL using the techniques shown in the following test agent script:

```
# Import tells TestMaker where to find Tool objects
from com.pushtotest.tool.protocolhandler import \
ProtocolHandler, SOAPProtocol, SOAPBody, SOAPHeader

from com.pushtotest.tool.response import Response
from java.lang import Long

# Set the default values defining the location of the host
host = "examples.pushtotest.com"
port = 92
path = "axis/servlet/AxisServlet"
```

```
endpoint = host + ":" + str( port ) + "/" + path

# Create a SOAP object to communicate with the host
protocol = ProtocolHandler.getProtocol("soap")

# over an SSL encrypted connection
protocol.setScheme( "https" )

body = SOAPBody()
protocol.setBody(body)

# Set the endpoint values
protocol.setHost( host )
protocol.setPath( path )
protocol.setPort( port )

body.setTarget( "responder_rpc" )
body.setMethod( "Respond" )

body.addParameter( "wordcount", Long, 150, None )
body.addParameter( "delay", Long, 100, None )

print "Sending request to server..."
response = protocol.connect()

print "Here is the response:"
print response

print
print "Agent done."
```

With the exception of the `setScheme()` method, this is the same agent presented in earlier chapters. The `setScheme()` method tells the `SOAPProtocol` object referenced by the protocol variable to use an SSL connection to communicate with the host.

.NET Passport Authentication

Microsoft .NET Passport is a single sign-in system that enables developers to defer their user authentication to .NET Passport servers. (Chapter 9 introduces the Microsoft .NET framework for building Web-enabled applications.) To provide this service Microsoft uses a combination of keys and configuration files that ensure user identity.

Early .NET applications used SSL encryption to secure SOAP requests and responses between the application and Passport servers. Later in this chapter we will see why SSL is not very well suited to secure .NET-based Web-enabled applications. For now we will give a brief overview of the security technology Passport provides.

From the standpoint of a Web site implementing .NET Passport single sign-in, .NET Passport Manager is the main service interface. Passport Manager has traditionally been a Microsoft COM server that could be used from ASP. However, the .NET Framework implements most of the same functionality in managed code via the `PassportIdentity` class.

During authentication, .NET Passport Manager is responsible for providing a link to the .NET Passport servers with all appropriate data in the query string so that the .NET Passport servers can perform the appropriate validation and return the user to the suggested site. .NET Passport Manager uses a combination of input parameters, registry settings, and data stored on the .NET Passport servers about your site to give the .NET Passport servers the information they need to perform the authentication with the requirements of the site. When authentication is established, the user browser redirects to the site and cookies are set to hold the user's .NET Passport ticket and profile information for subsequent requests. For details on Passport, go to http://msdn.microsoft.com/webservices/ and http://msdn.microsoft.com/downloads/default.asp?url=/downloads/sample.asp?url=/MSDN-FILES/027/001/644/msdncompositedoc.xml&frame=true.

HTTP Basic Authentication

The HTTP protocol defines a basic authentication method that provides very simple security to protect the results of an HTTP request. In an HTTP/HTML environment—as described in Chapter 6—a Web host optionally requires the HTTP header to contain user id and password entries to submit the request.

If the HTTP headers are missing, then the server responds with a 401 Unauthorized error code. Rather than display the 401 Unauthorized error code, most browsers display an alert that asks the users to type in their id and password. The browser then submits the same request plus the additional HTTP headers containing the id and password.

TestMaker supports HTTP basic authentication in its `HTTPProtocolHandler` object. In the following excerpt of an intelligent test agent implemented in TestMaker, the `http.setUsername()` and `http.setPassword()`

methods define the id and password to be sent to the host in the HTTP header.

```
from com.pushtotest.tool.protocolhandler import ProtocolHan-
dler, Body
http = ProtocolHandler.getProtocol("http")
http.setUsername("MyName")
http.setPassword("MyPassword")
...
http.connect()
```

HTTP Basic Authentication should be used only for the least security risks. HTTP Basic Authentication expects to find a user id and password in the HTTP header of a request. The header information moves across the Internet in clear text so anyone with a network monitor can see the unencrypted id and password information.

SOAP and Security

Most first-generation Web-enabled applications were deployed in internal integration projects behind a company's firewall. However, some companies are now deploying Web-enabled applications to expose internal systems over the Internet to business partners, vendors, and customers. Early adopters of Web-enabled applications technology can be found in the financial, government, and healthcare sectors, where risk of attack is greater as the data exchanged is often sensitive or high-value in nature.

SSL is effective to encrypt and authenticate SOAP-based Web-enabled application communication in first-generation Web-enabled applications. These early services typically move messages only between two points. SSL does fine in this environment. The SOAP client obtains a copy of the server certificate, authenticates the server, and establishes an encrypted channel.

The problem with SSL appears in B2B transactions. SOAP interfaces of Web-enabled applications access advanced functionality that business-to-consumer Web-enabled applications would not be able to access. For example, a Web site in a B2C application does not need the client certificate to carry out a sale of a low-priced item like a videotape or a pillowcase. In a B2B environment where the risk of making a mistake is higher, the SOAP host is more concerned about authenticating the client.

Client-side SSL certificates are used when Web-enabled applications require two-way authentication. The clients authenticate the servers and vice versa. The downside comes in the form of a management nightmare: every client and every server needs a digital certificate.

Extranets using client-side SSL are widely deployed to provide authorization but the localized user management demanded by client-side SSL is anathema to the hands-off and loose integration solution offered by SOAP-based Web-enabled applications. Even worse is the broken promise that Web-enabled applications enable organizations to communicate with each other without having to agree on common packages or carry out extensive integration. SOAP and WSDL were meant to be deployed quickly to enable loosely coupled systems. Then, SSL comes along demanding client-side SSL certificates distributed to every SOAP-based Web-enabled application that will make a request of our Web-enabled application. Without a solution there is going to be a hanging in town soon!

An old boss of mine used to remark: "What we don't know we really don't know." He meant that when one begins working with new technology, one rarely knows even the right questions to ask. This saying applies to securing SOAP-based Web-enabled applications. Future hacks, attacks, and the mistakes with Web-enabled applications are guaranteed. Our job today as developers, QA technicians, and IT managers is to look inwards and outwards in the following areas:

- Look inwards at the weakness in our systems. Then apply monitors and tests to verify when those weaknesses are being exploited.
- Look outwards to see which technologies will be applied over the coming year that will solve security weaknesses.

For example, looking inwards we can see that SSL does not protect Web-enabled applications from more traditional forms of attack. Web servers around the world are attacked daily with buffer overflow attacks. A hacker could stage a buffer overflow attack by sending through parameters that are longer than the Web-enabled application expects. Validating the content of messages before routing to the Web-enabled application counteracts this type of attack.

Looking outwards, several new message-level security technologies can improve SOAP-based Web-enabled application security in addition to SSL.

These XML security technologies play in the application or message layer of the OSI stack.

- **XML Schema** describes the allowed content and structure of an XML document. In a security light, XML Schema ensures that SOAP documents contain no malicious data. For details, go to http://www.w3.org/XML/Schema.
- **XML Signature** verifies an XML document was not changed between the client and service provider. An XML Signature includes an X.509 certificate, which links the requester with the request. An XML Signature uses the X.509 certificate within the signature to authenticate the requester. This involves validating the status of the certificate, which can be achieved using XKMS. For details, go to http://www.w3.org/Signature/.
- **The XML Key Management Specification** (XKMS) provides an XML interface to PKI services, including key distribution, validation, and revocation. XKMS enables a security checking service where one Web-enabled application can authenticate another's X.509 certificate. For details, go to http://www.w3.org/2001/XKMS/.
- **SAML** enables authentication and authorization information to be shared by multiple parties across security domains. With SAML, a SOAP-based Web-enabled application makes XML requests for authentication and authorization information. A request may contain an assertion signed by a trusted SAML authority that can be used by the service provider to authorize usage of the Web-enabled application. The access management products industry appears to be getting behind SAML to ensure that they can share authorization information. For details, go to http://www.oasis-open.org/committees/security/. My article, *SAML Myths and Misunderstandings,* is available at http://docs.pushtotest.com.
- **Liberty Alliance** is a standards body with Microsoft's strong support that is working on a single sign-in standard for groups of service providers that wish to federate their sign-in process. Details are at http://www.projectliberty.org/.
- **Web-enabled application Interoperability (WS-I)** is a standards body working on definition of a second-generation

Web-enabled applications-based platform, including Web-enabled application security. Details are at http://www.ws-i.org/.

The excellent thing about these technologies is that they are designed to be used in combination to protect Web-enabled applications from unauthorized access.

The second wave of Web-enabled applications is shaping up now and looks to be even more needing of message-level security than ever. Chapter 4 on SOAP predicts a future where groups of Web-enabled applications provide functions for a much larger application. This "federated application architecture" expands the need for security technology and intelligent test agents to verify Web-enabled application scalability and performance in secure environments. See http://docs.pushtotest.com for articles on second-generation Web-enabled application security technologies, including an article on the *Myths and Misunderstandings Surrounding SAML*.

Generating Certificates and KeyStores

While public key security technology exists and is well proven, most people I know do not want to even think about certificates in their everyday life. This behavior has led to the browser publishers including an arbitrary set of root certificates that they trust. When you browse a Web site that does not have a matching certificate, then the browser pops up a dialog message asking if you want to accept the unknown certificate.

The TestMaker framework for building intelligent test agents is a Java application that uses the JSSE library to provide SSL services. JSSE comes with Verisign and Thawte X.509 certificates installed. When JSSE connects to a server that has no matching certificate, it throws an exception.

```
URL exception. javax.net.ssl.SSLException: untrusted server
cert chain
```

This section presents the experiences of the U.S. Navy working TestMaker test agents and secure hosts. Then, we finish this chapter with instructions on using the Java keytool utility to use your own digital certificates.

Navy Supply Information Systems Activity (NAVSISA) Experiences

We were requested to assist in stress testing a Web server belonging to one of the Navy activities here at our base. They suggested using Test-Maker. The site was not secure, i.e., the URL scheme is "http." The page we had to drive was a simple one consisting of only one text field. It was an easy thing to do using TestMaker and we were very successful.

Others heard of our success and asked us to help them also. However, this new request was for a secure site, i.e., the URL scheme is "https." Our first attempt yielded this message:

```
"ElementURL: URL exception. javax.net.ssl.SSLException: untrusted
server cert chain"
```

Our problem was that the server was using Department of Defense (DoD) certificates that did not come with TestMaker.

We asked the TestMaker support team for help and they suggested some sources of information relating to how to put our DoD certificate chain into the keystore used by the Java VM using "keytool."

We downloaded our CA certificates (dodroot.cac) file and tried to import it into the "cacerts" file used by our Java VM and it wouldn't work. I think the keytool doesn't like the format the DoD uses (PKCS#12). So, we imported "dodroot.cac" into our browser (Microsoft Internet Explorer 5.5) and then exported it into an X.509 format file, which keytool would accept.

We then imported the X.509 format "dodroot.cac" certificates into our "cacerts" keystore using the keytool import command.

At this point, we were free of the "untrusted server certificate" error message mentioned above—but we had other problems.

The Web site we were currently testing required authentication by the application itself. The first page that was presented to the user had two text fields that the user had to fill in with their user-name and password. We didn't know what to put into the URL that we construct within our TestMaker script to do the authentication. We used a tool from HP (HP OpenView Internet Services) to discover that the keywords we needed were "USERID" and "PASSWORD." So once again we succeeded.

Next came a request to test a Web site that was also secure but used the server for authentication instead of the application, i.e., the server was configured to tell the browser to pop up a dialog to request the user credentials instead of the application asking for them.

This method of authentication makes use of the HTTP protocol directly instead of some part of the URL as was the case before. In order to do this kind of authentication, we needed some way to influence how TestMaker would construct the HTTP headers. In the documentation, we found the answer: the `setUsername()` and `setPassword()` methods for the `HTTPProtocol` object would supply the key/value for the HTTP header we were trying to get TestMaker to construct. The actual Test-Maker statement was:

```
from com.pushtotest.tool.protocolhandler import \
ProtocolHandler, Body
http = ProtocolHandler.getProtocol("http")
http.setHeader("Authorization", \
"Basic dHQwMDAwMDAwMDE6c3dtMTIzNDU=")
...
http.connect()
```

Part of the way we figured this out was to consult the RFC for HTTP 1.1 (RFC2068 section 11.1, Basic Authentication Scheme) that is found on http://www.w3c.org. We also used a network analyzer/sniffer to see what the client actually sent out when communicating with the server.

The "funny stuff" following the word "Basic" is the userid:password encoded using the so-called base64-encoding scheme. Naturally, we had no such capability so we had to go to the Web for a base64 encoder program.

If we may be of service to others, we'll try to help.

Ray Leiter

Raymond_R_Leiter@fmso.navy.mil

The Java Keytool

The TestMaker framework for building intelligent test agents is a Java application that uses the JSSE library to provide SSL services. JSSE comes with Verisign and Thawte X.509 certificates installed. So test agents may directly connect to servers that use Verisign and Thawte certificates. When a test agent establishes a secure connection to a server with another type of certificate, an exception is thrown:

```
URL exception. javax.net.ssl.SSLException: untrusted server
cert chain
```

In this section I show how to use the built-in Java keytool utility to use new digital certificates, generate keys, keystores, and truststores. For details on keytool, go to http://java.sun.com/docs/books/tutorial/security1.2/index.html.

In the following text, we step through several commands to create a self-signed certificate for the server and the client and store them in keystores server.keystore and client.keystore. The commands also export the server certificate to a file named server.cer that is then imported to the client keystore named client.keystore. The client's certificate is exported to file client.cer and imported to server's keystore server.keystore.

By taking these steps, the keystore client.keystore can also be used by the client as the truststore to verify the certificate supplied by the server and vice versa.

Note that the character \ used in the following commands indicates continuation of the same line. The \ character is not in the actual commands.

Begin by generating the Server KeyStore in file server.keystore with the following command.

```
java_home\bin\keytool -genkey -alias tomcat-sv \
-dname "CN=localhost, OU=X, O=Y, L=Z, S=XY, C=YZ" \
-keyalg RSA -keypass changeit -storepass changeit \
-keystore server.keystore
```

To export the certificate from keystore to an external file server.cer, use the following keytool command:

```
java_home\bin\keytool -export -alias tomcat-sv \
-storepass changeit -file server.cer \
-keystore server.keystore
```

To generate the Client KeyStore in the file client.keystore, use the following keytool command:

```
java_home\bin\keytool -genkey -alias tomcat-cl \
-dname "CN=Client, OU=X, O=Y, L=Z, S=XY, C=YZ" \
-keyalg RSA -keypass changeit -storepass changeit \
-keystore client.keystore
```

To export the certificate from the keystore to an external file named client.cer, use the following keytool command:

```
java_home\bin\keytool -export -alias tomcat-cl \
-storepass changeit -file client.cer \
-keystore client.keystore
```

To import the client's certificate into the Server's keystore, use the following keytool command:

```
java_home\bin\keytool -import -v -trustcacerts \
-alias tomcat -file server.cer -keystore client.keystore \
-keypass changeit -storepass changeit
```

Lastly, to import the server's certificate into the client's keystore, use this keytool command:

```
java_home\bin\keytool -import -v -trustcacerts \
-alias tomcat -file client.cer -keystore server.keystore \
-keypass changeit -storepass changeit
```

These steps accomplish a number of keytool functions. Keep the following in mind when using keytool commands:

- The server name specified in the server's certificate is localhost. If you plan to access the server with its symbolic name or IP address on the network, you must specify that string while generating the key. You can also have multiple certificates for the same physical machine.
- Though this script is sufficient for running the example, a real production deployment must use a proper distinguished name value, getting certificates signed by an authorized CA.

- I highly recommend using a production grade tool to manage client certificates, especially if the server wants to authenticate the client based on the supplied certificate.

This chapter did not presume an understanding of the underlying concepts of Public Key Infrastructure or SSL. We recommend *Java Security* by Scott Oaks, published by O'Reilly & Associates. For details, go to http://security.oreilly.com/.

Summary

This chapter provided tips and techniques to test Web-enabled applications in secure environments. The use of SSL technology enables businesses and organizations to secure the communication between Web-enabled application client and host. While businesses have largely adopted PKI as the unofficial standard for securing Web-enabled applications, this chapter also pointed out the needed new XML technologies to manage certificates, access, and authorization that SSL does not provide.

In the next chapter we observe what happens after intelligent test agents are written and used to test a system, including answering questions about the needed equipment, network bandwidth, and resource requirements to make a test produce meaningful results.

11

A Web Application Framework from Construction to Test

The previous chapters show the techniques an intelligent test agent needs to use to communicate and command a Web-enabled application. This chapter takes the next step. The techniques and examples in this chapter show how to configure and run a concurrency and scalability test. Then, the next chapter shows how to analyze the results.

The Trading Desk and Intelligent Test Agents

Television, films, and newspapers often feature pictures of stock and bond traders working in front of computer and telephone systems. This image looks similar to other high-volume, rapid-fire settings, such as airline flight controllers, emergency dispatchers, and computer network operators. A large panel gives the trader information about the market, traders use a keyboard to order stocks to be bought or sold, and the phone barks out instructions from nervous investors. When it works well everyone makes money. When the system slows or stops, careers are endangered. Bring in the intelligent test agents!

Intelligent test agents are an ideal tool to check scalability and performance and to monitor the service for uptime in high volume, high capacity near real-time systems. These systems have become increasingly complex of late. For example, online stock brokers often feature services to tie buy and sell instructions to market conditions. One order to sell shares in an energy company may be triggered when the price of oil falls below a certain amount. To accomplish such functions the broker's system must autonomously gather oil prices from one system and instruct a buy/sell system to sell shares. How do you test this system?

The complexity built into modern interoperable Web service-based systems requires new test methodologies and techniques. For example, the trading system would use a test agent to simulate the complex market conditions that trigger a Web service action. This chapter shows how intelligent test agents that use multiple protocols and data sources produce meaningful test results.

Scalability Test Goals

Businesses today that rely on complex interoperating systems need test methodologies and techniques to answer basic operating questions. In a stock trading business, the system operator's goals are achieved as a measurement of system up-time and speed to conduct transactions. While there are many aspects to the stock trader's system that may be tested, in this example the most important tests answer the following questions:

Table 11–1 Establishing the Method and Measurement for a Test

Issue	Method	Measurement
Do I have the right number of servers?	Each part of the system contributes overhead to processing transactions. In a trading system, firewalls, routers, load balancers, Web servers, application servers, database servers, and storage systems each add overhead to process a transaction. A system test uncovers the largest bottleneck to process transactions. Resolving system bottlenecks to the anticipated number of users answers the issue of equipment purchases to forecasted demand.	Measure round-trip transactions-per-second timing from the client machine. Then, measure overhead from each major system component while processing a transaction.

Table 11–1 Establishing the Method and Measurement for a Test

How does the system perform under increasing loads of use?	System users have a built-in capacity to tolerate some slow system performance. However, there quickly comes a point where slow system performance is counterproductive to the businesses' goals. User productivity should be measured by interviewing users regularly to determine improvements to system performance and by learning management's productivity goals per user.	Measure the time it takes to accomplish tasks on the system; also measure the subtasks needed to accomplish each larger task.
Will enough activity eventually cripple the system?	Complex interoperating systems often run out of system resources over time. Memory leaks, unreleased file handles, too little hard drive space, and the random hardware problem can grind system performance into the ground. Simulating large concurrent usage over a period of days uncovers system resource problems during the test.	Measure system resources (CPU time used, memory allocations, disk usage) during an extended test of 1–2 days.

The test agents presented in this chapter aim to answer these issues for a fictitious stock trading company. We construct the test by first looking at the existing system infrastructure and identifying the user archetypes.

System Infrastructure

For our fictitious stock trading company we define good system performance as the following:

- The service works just as well under the stress of one user as 100 concurrent users—assuming the users are a mix of the users, including traders, managers, analysts, and network managers.
- All users have good experiences even though some users operate a single function while other groups use all the functions.
- The Web service is available whenever a user shows up.

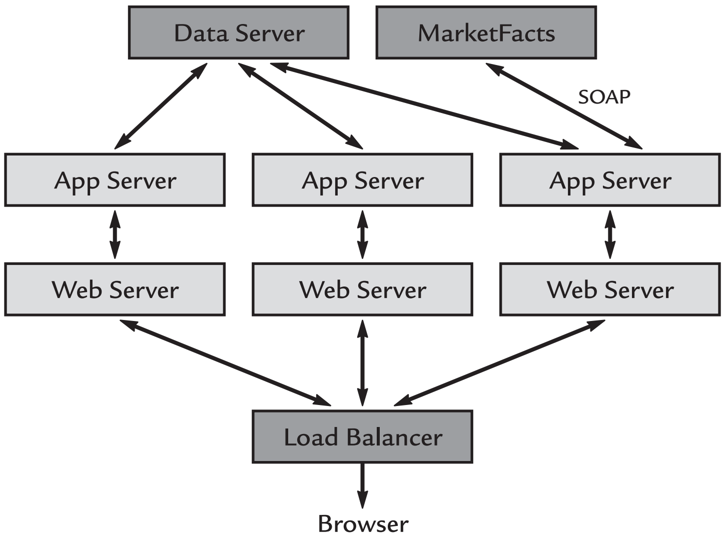


Figure 11–1 The Flapjacks system architecture implemented by a fictitious stock trading company. The layer of Web servers acts as intermediaries to handle simple page requests. The application server handles the complex procedures needed to dynamically assemble requested information.

Choosing the right components to a system that hosts a Web service greatly determines how likely the service will achieve its goals. Earlier chapters presented a framework for building and deploying Web-enabled applications called the Flapjacks architecture. As Figure 11–1 shows, the Flapjacks architecture that I introduced in Chapter 2 in a stock trading company deploys many small servers that are accessed through a load balancer, providing a front-end to a powerful database server.

The stock trading company system puts the core of the business logic into a tier of application servers. User requests are served through a load balancer and Web server. The trading application software accesses user accounts and trading history from the database server. The system accesses current market conditions through a SOAP connection to an external MarketFacts service.

The stock trading company uses the Flapjacks architecture to minimize downtime and provide flexibility in changing the number of Web and application servers over time. Additionally, the Flapjacks architecture is testable.

Figure 11–2 shows intelligent test agents added to the system infrastructure to record, measure, and in some cases drive the system components. In the stock trading company example, the business goals for the test require the agents to

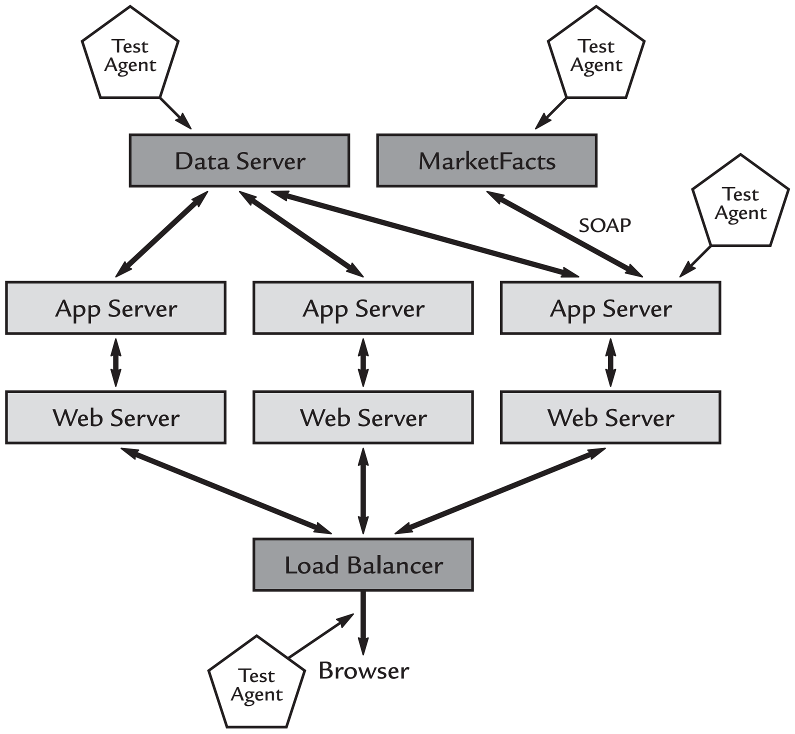


Figure 11–2 Intelligent test agents added to the Flapjacks infrastructure equip the system to determine bottlenecks and component-level performance and scalability.

monitor the major components, simulate user requests, and check performance and availability of remote resources, such as the MarketFacts service.

Each test agent monitors a different component, as follows:

- The test agent monitoring the data server records CPU and hard disk utilization of a database server application.
- The test agent monitoring the application server records the time it takes to process transactions.
- The test agent checking the MarketFacts service determines network latency overhead, Web service availability, and transaction duration.
- Lastly, the bottom-most test agent simulates the actions of a set of user archetypes.

Next, we learn how to identify the typical user activities on the system by identifying a few user archetypes.

User Archetypes

The stock trading company is a good case to apply user archetypes. User archetypes are a powerful technique to model users in a multifaceted system that require multiple steps to accomplish a goal. See Chapter 1 for details on user archetypes.

User archetypes are a good technique to overcome the paralyzing fear many developers go through when they consider how their software will be used. Many developers think testing must cover a general cross-section of the user community. Other developers believe high-quality software is tested against the original design goals of a Web-enabled application as defined by a product manager, project marketer, or lead developer. These generalized testing approaches usually miss the target by a long shot.

General testing makes large assumptions of how the aggregate group of users will use the Web-enabled application and the steps they take as a group to accomplish their common goal. Systems simply are not used this way. In reality, each user has his or her own goal and method of using the system to achieve this goal.

The antidote to general testing is to develop and apply *user archetypes*. Archetypes are prototypical users based on the real people that will use the system. The best way to build a user archetype is to start with a single user. Choose just one user, watch them use the Web-enabled application, learn their goals, and watch the steps they take to achieve their goals. The better you understand an individual user's needs, the more valuable your test agent will be.

Many developers have taken the archetypal user method to heart. They name their archetypes and describe their backgrounds and habits. They give depth to the archetype so the rest of the development team can understand the archetype better. Consequently, the team understands the test better because they can relate to the user archetype. They can also use archetypes when describing problems in the Web-enabled application. Consider these user archetypes for the stock trading company:

- Mira—stock trader, works the trading desk by taking orders by phone and buying or selling stocks through the system terminals. Mira is a compulsive person: After learning to play golf she took a three-week unpaid vacation to follow Tiger Woods around golf courses in six states. Mira was married and is now divorced, with no children. A dark circle from her daily Starbucks coffee has developed in one spot on her desk. She explodes with anger and sarcasm when things don't go her way.

- Simon—trainee stock trader, just graduated from business school at a prestigious university; this is his first job out of school. Simon was on the rowing team, got decent grades despite much partying, and has a new girlfriend. Simon works the regular shift at the trading desk and is studying for a state-sponsored stock analysis certificate that he may earn in two years.
- Doree—trading operations manager, started as a trader 10 years ago when she started at the firm. She outperformed all other traders while still finding time to train the new traders. Doree got married two years ago. Her husband is a sales executive in a competing firm. Doree is an indoor person during summer and loves skiing, sledding, and snowshoe trekking during the winter.
- Vicki—systems manager, friend of the CEO's son. Started at the firm by fixing the CEO's desktop computer system two years ago. Since then Vicki has taken on increasingly complex systems management roles. Vicki is soft spoken and at times timid, a trait she gets from her parents who are also Buddhist and anti-war pacifists. Vicki's worst nightmare is to arrive at the office to face Doree and Mira's fury.

These four user archetypes do not cover 100% of the users of the system. Instead they identify 100% of these four user archetypes goals. As time goes on, additional user archetypes will be defined to model more user goals and more test agents.

Developers who use Universal Markup Language (UML) techniques may find a user archetype similar to UML actors. User archetypes are an in-depth idea of a UML actor. In this case, you are able to "role play" the actor into a real person.

User archetypes facilitate developing intelligent test agents. For example, modeling a test agent against Mira's archetype concentrates on the speed of transactions. Mira needs to retrieve stock information quickly, read a few facts, and then place an order. To accomplish these goals, the test agent signs in to the system, uses the search function to find a stock, and then places an order. The agent takes no delay time between steps as it tries to complete as many transactions as fast as possible. The test agent takes several steps to accomplish its goals and highlights when the system does not perform to Mira's speed goals.

A test agent modeled to achieve Vicki's goals concentrates on accessing and monitoring the system. Vicki needs to know when the system is not performing well. She wants to know before Mira and Doree find the system is sluggish or slow. The test agent needs to query the system's performance logs to learn which transactions are taking the longest time.

Understanding the archetypes is the key to making the test agents intelligent. For example, a test agent for Vicki may behave more persistently than a test agent for Simon. If a test agent tries to execute a trade that fails the test agent, Vicki would try again. Archetype behavior makes a test agent intelligent.

Using archetypes to describe a user is more efficient and more accurate than making broad generalizations about the nature of a Web-enabled application's users. Archetypes make it easier to develop test agents modeled after each user's individual goals.

Next, we investigate the requirements to conduct a test of the stock trading example. Then we will construct the intelligent test agents to run the test.

Understanding the Test Requirements

At one time in the history of software development, testing efforts were organized and conducted by a single person. Since then we have found that testing complex interoperable information systems, especially Web-enabled applications, demands the effort and cooperation of a number of teams. It is no longer a one-person task. For example, in the stock trading company example, just imagine the effect of 1,000 concurrently running test agents modeled after Simon. Simon conducts research before most stock trades. The research causes network traffic and load on the local datacenter and to the external MarketFacts service. Without enough planning and coordination, such a test might cause the real Vicki—who manages the stock trading company network—to suspect that the network is under attack from a hacker. Also, the MarketFacts sales manager may expect that the stock trading company's next monthly bill will be huge because the number of requests for market data is skyrocketing, when in fact the data is in response to requests from a test agent.

Coordinating the efforts of many people during a test requires an understanding of their needs. We can state the needs as a list of requirements. In the example stock trading company's case, the requirements are stated in Table 11–2.

Table 11–2 Requirements to Test the Stock Trading System

Requirement	Why is this important?	Solution
Tests must be configurable	Developers, QA analysts, and IT managers each want to run the tests on their own. The test agents must be configurable without requiring the original test agent author's intervention.	Test agents use configuration files to define test parameters.
Tests must be reusable	Any developer, QA analyst, or IT manager may build a test agent. Once built the test agent is an asset to the company that anyone can use.	The test agents are all developed in the same framework, which includes an easy-to-use scripting language, protocol handler object library, and programmatic launcher. The agents are packaged into shareable software applications that may be checked in to a source code control system for distribution.
Tests must be maintainable	Even if a test agent only contains a single line of code, it will eventually require changes, updates, and modifications to remain useful.	The agents are packaged into shareable software applications that may be checked in to a source code version control system for distribution.
Tests must be modular	Test agents implement functions that will be useful in a larger scope of software development automation. For example, an automatic build-and-deploy system would benefit from having a test agent plug-in to perform build–deploy–test functions.	Test agents are callable from external applications. Additionally, test agent results may be shared with the external.

The requirements are only asking for the test agents to be configurable, reusable, and maintainable and module. What is so hard about that? Well, plenty, as we will see when we construct the stock trading company test in the next section.

Constructing the Test

The test agents presented in this chapter answer basic questions about the reliability, scalability, and functionality of a fictitious stock trading company. Earlier, we learned how to define the goals of the test, how to use user archetypes to model test agent behavior after the goals of the users, and how to define the requirements for the test environment. Now, we will construct the test.

First, let's understand the flow of the test agent environment. Figure 11–3 shows the major components of the test and how they relate to each other. We construct the test in three successive parts: setup, run, and analysis. In practice, separating these parts provides an easy way to manage and maintain tests over time. Each part factors into a base function that contributes to the overall success of the test.

While Figure 11–3 shows the flow from one part of the test to the next, an understanding of how the component parts are designed is important to understanding the test components. The stock trading company test is implemented with the following steps:

- A master agent calls agent modules to do the actual work of driving the system under test.
- The agent modules implement a behavior defined in the user archetype descriptions. For example, *doree.a* is a TestMaker script that implements the behavior of the Doree user archetype. Test results are stored in a special delimited data file.
- A tally module that analyzes the contents of the delimited results data file and presents findings.

Figure 11–4 illustrates the master component. In the stock trading company system test there is little setup needed so this test implements the setup functions in the master component. That is because the test agents for this test expect that the user accounts for Doree, Mira, Simon, and Vicki are already established and that the system can handle trading orders and

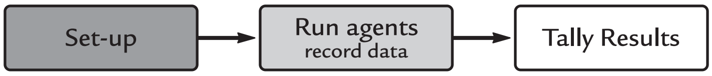


Figure 11–3 The overall test is divided into three parts. The setup part describes the test environment, number of concurrently running test agents, and behavior of each agent based on a user archetype. The second part actually runs the agents and records the results to a log file. The third part analyzes the recorded results and presents the findings.

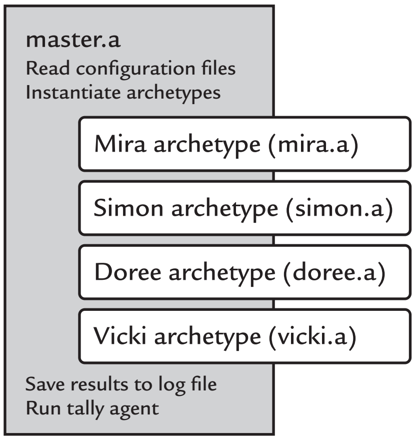


Figure 11–4 The *master* agent encapsulates the entire test. Results are logged to a file that are summarized and presented with the *tally* agent.

requests for information without doing preconfiguration prior to the test. The system's state is expected to be configured before the test agents run.

In other system tests, configuring the system for a test requires the system state to be configured. That includes operations such as establishing new user accounts, setting usage parameters for each of the new accounts, and entering the market information that the test agents will query. If the system depends on some sort of preconfiguration, then the test agent is an ideal form to document and configure the setup. This includes creating as much of the initial state as possible.

In addition to setup, the master component instantiates the user archetype behavior in a set of concurrently running threads. Each thread encapsulates the behavior of a user archetype and issues the commands needed to drive the stock trading system. The configuration files for the test tell the master component how many threads to create and what type of user archetype

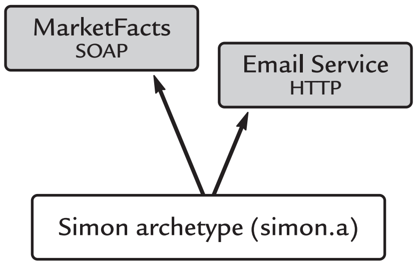


Figure 11–5 The test agent modeled after Simon follows market trends using SOAP interfaces to the MarketFacts service and also sends email alerts via a Web-based email service.

behavior to use in each thread. The actual user archetype behavior is implemented in a set of files that the master component calls. For example, the mira.a file contains the implementation of the Mira user archetype behavior. The same is true for Simon, Doree, and Vicki.

Looking at a single user archetype implementation shows the basic function of the behavior. Figure 11–5 shows the Simon user archetype. The description of the Simon user archetype earlier in this chapter showed that he is a trainee stock trader that conducts stock trades. Simon is also interested in trends in the market so he also makes several requests to the MarketFacts service while he places stock trade orders. When Simon detects a trend underway, he alerts his fellow traders Mira and Doree. Simon sends email messages with the trend he tracks.

Later in this chapter we discover how the Simon agent is coded. For now, let us examine how to define the test behavior in terms of the steps the user archetypes take to accomplish their goals.

In the case of Mira, the user archetype describes user behavior that is both aggressive and rapid in nature. To capture Mira's behavior, the test agent performs the following actions:

- It tries to process several stock trades at the same time.
- It makes one request for a stock price after another. If any one request takes more than two seconds, the agent cancels the request and makes another.
- The agent makes a burst of requests and then pauses for a few moments, as though the agent is Mira taking a new phone call that requires her to listen for a moment before acting.

- It checks for email messages sent from the Simon test agent. If email messages exist, then it reads them.

The Simon user archetype describes behavior that is more evenly paced than Mira. Simon makes many requests to the MarketFacts service to understand the market conditions. The following are the actions the Simon test agent takes:

- It makes a request for two to five reports from MarketFacts. It repeats this after spending a moment to analyze the contents of the reports.
- It occasionally makes a stock trade.
- It sends an email message to Mira and Doree with the results of research.

The test agents that implement Doree's behavior can be just as energetic as Mira's. However, Doree mostly has a polling job where she watches and makes sure the trades are being handled efficiently, as follows:

- A Doree test agent checks for new messages over and over again. This occurs repeatedly from when the agent starts running until the test is done.
- Once an hour the agent posts a new message to the traders indicating market trends and stocks the firm wants to push.
- Every 10 minutes the agent requests a management report showing hour-to-date sales for each broker.

The master component uses the configuration files to learn the total number of threads to create and the relative amount of user archetypes to use. For example, suppose the configuration file calls for 100 concurrently running threads. The configuration file defines that 35% of the threads use the Mira user archetype, 20% Simon, 30% Doree, and 15% Vicki. Figure 11–6 graphically shows the split among user archetypes.

Now that we understand the actions of the user archetypes and the overall construction of the test suite, we will learn to write the code to implement the test.

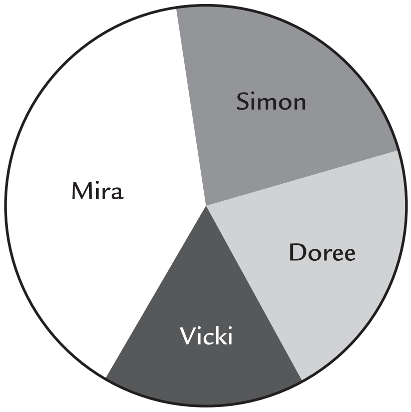


Figure 11–6 The master component uses the configuration files to determine the mix of user archetype behaviors it will create in the overall group of threads. The total number of agents may increase, but the percentage mix of user archetypes remains constant.

Implementing User Archetypes in Code Modules

Next, we look into how the Stock Trading Company test agents are crafted. To get hands-on I present a TestMaker test agent that implements the user behavior in a test. TestMaker is described in Chapter 4. We begin by showing how the Doree user archetype is implemented as an intelligent test agent.

Please consider two things when looking at the following code. First, the complete code to the stock trading test suite is found at http://thebook.pushtotest.com. Second, the functions implemented for this test agent are merely simulations of the actual calls an agent would make for a real test.

The Doree agent checks for new messages repeatedly. Every once in a while, the Doree agent posts an announcement message to the rest of the traders. Following are script functions that implement these two methods. The first function defines a method that checks the company mail server for new messages. Here is the complete method, followed by a detailed explanation. All of the code presented in this book is also available for download at http://www.pushtotest.com/ptt/thebook.html.

```
#-------------------------------------------------
# Read email messages

def read_emails():
    protocol2 = ProtocolHandler.getProtocol("mail")

    protocol2.setHost("mail.pushtotest.com")
    protocol2.setUsername("buddy")
    protocol2.setPassword( "email" )

    response = protocol2.connect()
    response.setPermission( Folder.READ_WRITE )
    response.setFolder( "INBOX" )

    messages = response.getMessages()

    for i in messages:
        print i.getFrom()[0], i.getSubject()
        i.setFlag( Flags.Flag.DELETED, 1 )

    response.close( 1 )
```

The `read_emails` method checks the mail server for new messages. When messages are waiting, the method downloads and deletes the messages. Next, we examine the individual commands in the `read_emails` method.

```
#-------------------------------------------------
# Read email messages

def read_emails():
    protocol2 = ProtocolHandler.getProtocol("mail")
```

First, we define a new method titled `read_emails` that we can call later in the same agent script. Then, the script creates a new instance of the `Mail-Protocol` object that we reference in the protocol variable. See the testmaker_home/agents/Mail_agent.a for information on the Mail protocol handler.

```
    protocol2.setHost("mail.pushtotest.com")
    protocol2.setUsername("buddy")
    protocol2.setPassword( "email" )

    response = protocol2.connect()
```

These commands tell the `MailProtocol` object where to find the email host and the account to use. PushToTest created the buddy email account on mail.pushtotest.com to enable you to run the test agent. Please do not abuse this email account.

```
response.setPermission( Folder.READ_WRITE )
```

The `setPermission` method instructs the mail host of our test agent's intentions. By default, `Folder.READ_ONLY` tells the mail host we will be reading email messages but not deleting them from the host. `Folder.READ_WRITE` tells the mail host to expect message delete commands.

```
response.setFolder( "INBOX" )

messages = response.getMessages()
```

The POP3 protocol uses a single folder titled INBOX by default. The IMAP protocol supports multiple folders.

Then, the code iterates through each message. The `getFrom`, `getSubject`, and `writeTo` methods return the from, subject, and message body.

```
    for i in messages:
        print i.getFrom()[0], i.getSubject()
```

The message body is also available from the `writeTo` method that takes an `outputStream` object as its input. The `setFlag` method enables the agent script to mark the email message for deletion from the mail host. The flag is set in this loop but does not actually happen until the `close()` method executes. The flag has no effect if the `setPermission` method is not set to `Folder.READ_WRITE`.

```
        i.setFlag( Flags.Flag.DELETED, 1 )
```

After we iterate through all the messages, the script closes the connection to the host. The `close` method takes a single boolean parameter. When the parameter is set to `True` or `1`, the `close` command instructs the mail host to delete any flagged messages. A `False` or `0` parameter tells the mail host to ignore the deleted messages.

```
    response.close( 1 )
```

Next, we see how the Doree agent posts an announcement message to the rest of the traders. Here is the complete method, followed by a detailed explanation.

```
#--------------------------------------------------
# Post a news alert

def post_news_alert():
    http_ph = ProtocolHandler.getProtocol("http")

    http_ph.setHost( "examples.pushtotest.com" )
    http_ph.setPath( "responder/htmlresponder" )

    http_ph.setType( HTTPProtocol.POST )

    body = ProtocolHandler.getBody( "http" )

    body.addParameter( "login", "myname" )
    body.addParameter( "passwd", "secret" )

    http_ph.setBody( body )

    response = http_ph.connect( 0 )
```

The `post_news_alert` method uses an HTTP form post to send a message. Next, we look at the individual commands that make up the Doree agent.

```
def post_news_alert():
```

First, we define a new method titled `post_news_alert` that we can call later in the same agent script. Next, the script creates a new instance of the `HTTPProtocol` object that we reference in the protocol variable. For information on the HTTP protocol handler, see testmaker_home/agents/signin_agent.a.

```
    http_ph = ProtocolHandler.getProtocol("http")

    http_ph.setHost( "examples.pushtotest.com" )

    http_ph.setPath( "responder/htmlresponder" )
```

The script instructs the HTTP protocol handler where to find the host and the servlet to post the message.

```
    http_ph.setType( HTTPProtocol.POST )
```

Next, the script tells the HTTP protocol handler that the request to the host will be in the form of an HTTP post command.

```
    body = ProtocolHandler.getBody( "http" )

    # send the sign-in values
    body.addParameter( "login", "myname" )
    body.addParameter( "passwd", "secret" )
```

The new body object holds the HTTP post parameters. We set the first parameter to have Doree's login identity and the second parameter to have her password.

```
    http_ph.setBody( body )
```

Then, the script tells the HTTP protocol handler to use the new body object.

```
response = http_ph.connect( 0 )
```

Finally, the script uses a special form of `connect()` that tells the HTTP protocol handler not to follow any redirect commands received from the host. Usually the response to a new message post is to redirect the user's browser to a "message posted" page. For the purpose of this test, we do not need to load the redirected page.

Functionally the Doree agent is ready to go. The `post_news_alert` and `read_emails` functions perform the test agent's actions. Next, we will see how the rest of the Doree agent is scripted to put these functions to work.

Here is the main section of the Doree agent. The complete code is listed followed by a detailed explanation.

```
#--------------------------------------------------
# Doree

def Doree( agent_num, local_name ):

    global running, log, up_count, record, tick, errortick

    random.seed( agent_num )

    up_count += 1

    pulse = Date().time     # record the start time
```

```
sleeptime = 0

news_end = Date().time + ( 10 * ( 60 * 1000 ) )

while running:

    status = "ok"
    datapulse = Date().time

    try:
        read_emails()

        if Date().time > news_end:
            post_news_alert()
            news_end = Date().time + \
            ( 10 * ( 60 * 1000 ) )

    except java.lang.Exception, ex:
        print "Error thrown:"
        print "  ",ex
        print
        status = "fail"
        errortick += 1
    except Exception, ex:
        print "Error thrown:"
        print "  ",ex
        print
        status = "fail"
        errortick += 1

    responsetime = Date().time - datapulse

    # Log the results to a comma-delimited file with
    # these fields:
    # response time, status, time index,
    # machine name, thread id number

    if record:
        totalresponsetime = Date().time - pulse
        tick += 1

        try:
            result = str( responsetime ) + "," + \
            status + "," + str( Date().time ) + "," \
            + local_name + "," + str( agent_num )
            print "result:",result
```

```
                log.log( result )
            except Exception, ex:
                print "Error while logging results: ",ex

        # This test agent thread sleeps after each request
        # to make the test more closely modeled to real
        # production environments where requests are
        # rarely uniform.

        time.sleep( random.randrange( 0, 5 ) )

    # We're done so reduce the thread count and exit
    up_count -= 1

#--------------------------------------------------
# Instantiate a thread

thread.start_new_thread( Doree, ( i, "Doree"+localname ) )
```

The Doree method implements the main part of the Doree agent. Next, we look at the individual commands in the `post_news_alert` method.

```
def Doree( agent_num, local_name ):

    global running, log, up_count, record, tick, errortick
```

The script defines the `Doree` method as the main part of the `Doree` agent. The `Doree` method expects to receive an agent number identifier and a `local_name` to use when logging test results. Additionally, the *master.a* script that instantiates the `Doree` agent sets up several global variables that the *Doree* script needs. These include the following:

- `running`—A flag that is set to true while the master component wants the Doree agent to run.
- `log`—The `log` object that Doree uses to log results data.
- `up_count`—A simple integer value that indicates how many copies of the Doree agent are running.
- `tick`—A temporary variable that tracks the number of successful transactions.
- `errortick`—A temporary variable that tracks the number of transactions that ended in error conditions.

```
    random.seed( agent_num )
```

The agent uses a random number generator to determine sleep durations after each transaction. We use the `agent_num` value as a seed.

```
up_count += 1
```

The master component uses threads to create multiple concurrently running copies of the Doree agent. The `up_count` global variable keeps track of how many agent threads are running. This is useful when shutting down a running test, as we will see later.

```
pulse = Date().time
sleeptime = 0
news_end = Date().time + ( 10 * ( 60 * 1000 ) )
```

Next, the script records the start time. This information is used later when logging test results. The `news_end` value is set to the current time plus 10 minutes (1000 milliseconds = 1 second).

```
while running:
```

The master.a component is at the control of the running variable. When the global variable `running` is set to true, the agents continue making requests. When master determines the test is over, it changes `running` to false and all the agents end their activity.

```
    status = "ok"
    datapulse = Date().time
```

The script uses the datapulse variable to record the time to process the transaction.

```
    try:
        read_emails()

        if Date().time > news_end:
            post_news_alert()
            news_end = Date().time + ( 10*(60*1000) )
```

The `try` command wraps around the calls to `read_emails` and `post_news_alert` and catches errors in the `except` block shown below. The `news_end` variable indicates when 10 minutes have passed. When 10 minutes are over, the script uses the `post_news_alert` method to send a message to the other traders.

```
        except java.lang.Exception, ex:
            print "Error thrown:"
            print "  ",ex
            print
            status = "fail"
            errortick += 1
        except Exception, ex:
            print "Error thrown:"
            print "  ",ex
            print
            status = "fail"
            errortick += 1
```

The except commands catch Python and Java errors that happen while processing a transaction. In the test agent example, both error types are handled the same way. The errortick value is incremented by 1 and the agent continues.

```
        responsetime = Date().time - datapulse

        if record:
            totalresponsetime = Date().time - pulse
            tick += 1

            try:
                result = str( responsetime ) + "," + \
                status + "," + str( Date().time ) + "," \
                + local_name + "," + str( agent_num )

                print "result:",result
                log.log( result )
            except Exception, ex:
                print "Error while logging results: ",ex
```

When a new test is started, the master component instantiates the user archetype threads. To smooth the impact of creating the new threads on the underlying Java Virtual Machine, the master component avoids instantiating all the threads at once. After all the threads are instantiated, the master component sets the record global variable to true to instruct all the agents to begin logging the transaction results.

The script logs the results to a comma-delimited file with the following fields:

- response—time in milliseconds it takes to complete each transaction. In Doree's case a transaction includes running the `read_emails` method and running the `post_news_alert` method every 10 minutes.
- Status—true indicates successful transaction, false indicates an error happened while processing the transaction.
- time index—the current system time (in milliseconds from 1904) when the transaction was started.
- machine name—the name of the local machine.
- thread id number—the assigned thread number for this agent.

```
        time.sleep( random.randrange( 0, 5 ) )
```

The Doree test agent sleeps after each request, as explained when we identified the user archetype earlier in this chapter.

```
thread.start_new_thread( Doree, ( i, "Doree"+localname ) )
```

The *master.a* component instantiates the Doree agent by using the `exec()` command to run the *doree.a* script file. The script file creates the `Doree()` method and then instantiates a new thread containing the `Doree()` method.

This section shows how to implement the Doree user archetype as an intelligent test agent. For details on how the Simon and Mira agents are constructed, look in at the *simon.a* and *mira.a* files in the code examples at http://thebook.pushtotest.com.

Next, we see how the master component is constructed to state the tests and instantiate the user archetype agents.

Implementing the Master Component

In the previous section, we covered the steps you need to follow to implement user archetype behavior in a script. Many references were made to the master component in the script comments. The master component is the conductor that stages, runs, and shuts down the test. You can find the master component of the Stock Trading Company test in the master.a file in the Code Examples at http://thebook.pushtotest.com.

This section shows how the master component is constructed. The master component is implemented in the following three parts:

- Set-up—runs the property.a script to initialize the global variables and creates the `log` object.
- Run—instantiates the test agents.
- Clean-up—stops the test agents and runs the *tally.a* script to tally the results.

Setup

Following is a listing of the entire setup section of the master component. After the script is a detailed explanation.

```
# Stock Trading Test Suite

# Import TestMaker TOOL and Java objects

from java.lang import Thread
import java
import time
from java.util import Date
from com.pushtotest.tool.logger import simplelogger
import sys
import thread

# Main body of agent

exec open( scriptpath + "properties.a" ).read()

running = 1
up_count = 0
record = 0
tick = 0
errortick = 0
first_time = Date().time

logtarget = resultspath + "results_" + localname + ".txt"

try:
    log = simplelogger.getInstance()
    log.setFileName( logtarget )
except java.lang.Exception, ex:
    print "Could not start logging. Shutting down agent."
    sys.exit()
```

The first part of the master component identifies the objects that are used in the script, initializes several global variables, and instantiates a new `log` object used to report the agent results.

```
from java.lang import Thread
import java
import time
from java.util import Date
from com.pushtotest.tool.logger import simplelogger
import sys
import thread
```

The `Import` command identifies the objects the script uses. TestMaker uses a version of Python called Jython. Jython is the Python language implemented entirely in Java. This approach's advantage is that Jython scripts can use Python objects and Java objects. The `Import` commands that identify `time`, `sys`, and `thread` are pointing to Python objects. The script also uses java.lang.Thread and java.util.Data objects.

```
exec open( scriptpath + "properties.a" ).read()
```

The `exec` command runs a script in another file. In this case, the `exec` command runs the script commands in the *properties.a* script file. *properties.a* sets several user-configured parameters that define the scope of the test.

```
running = 1
up_count = 0
record = 0
tick = 0
errortick = 0
```

The script commands establish global variables as follows:

- `running`—Tells agents to continue running.
- `up_count`—Counts the number of test threads that are up and running.
- `record`—Tells running agents when to start logging results data.
- `tick`—Keeps track of total completed transactions.
- `errortick`—Keeps track of errors.

```
first_time = Date().time
```

The script records the current time to use later when analyzing the results.

```
logtarget = resultspath + "results_" + localname + ".txt"

try:
    log = simplelogger.getInstance()
    log.setFileName( logtarget )
except java.lang.Exception, ex:
    print "Could not start logging. Shutting down agent."
    sys.exit()
```

Finally, the script instantiates a new `log` object for the agents to use to record their results. The `log` object is provided by the TestMaker TOOL package. The `log` object provides a simple, common way for the test agents to log their results to a common log file.

Run

The setup section of the master component establishes objects, global variables, and instantiates the `log` object. The run section instantiates the test agents to begin the test process. Next, we list the entire master component run section. Then, we explain this script. For the complete *master.a* script, look in the Code Examples directory that accompanies this book.

```
# Calculate how many of each type of agent to run based on the
# percentage value set in Properties.a

total = 0

for a in agents:
    total += a[1]

if total>100:
    print "Warning: The total number of agents specified"
    print "         in the Properties.a file is greater"
    print "         than 100%."

# Instantiate the agents as concurrently running threads for
at in agents:

    agenttype = at[0]
    agent_count = int ( ( float( at[1] ) / float( total ) )
* agentcount )
    for i in range( agent_count ):
        exec open( scriptpath + agenttype + ".a" ).read()
        time.sleep( startupdelay )

print "Threads started, we're testing!"

record = 1

start_time = Date().time
end_time = start_time + timetorun
while Date().time < end_time:
    time.sleep( 1 )

record = 0  # Tells running agents to stop logging results
recording_time = Date().time - start_time
```

This part instantiates the test agent threads, waits for them to test the system, and then records the start and end time. Following is the same code with a detailed explanation.

```
total = 0
for a in agents:
    total += a[1]
if total>100:
    print "Warning: The total number of agents specified"
    print "         in the Properties.a file is greater"
    print "         than 100%."
```

This first part calculates how many of each type of agent to run based on the percentage value set in the *properties.a* file. `agents` is a list object that contains a set of tuples. For example, `agents` may be set to:

```
["doree", 50]
["mira", 20]
["simon", 30]
```

The first value of each tuple identifies by name the user archetype to use for a test agent and the second value indicates the percentage mix of this type of test agent to use when testing. For example, [“doree”, 50] indicates that 50% of the total number of concurrently running agents will be using the doree user archetype behavior.

```
for at in agents:
    agenttype = at[0]
    agent_count = int ( ( float( at[1] ) \
    / float( total ) ) * agentcount )
    for i in range( agent_count ):
        exec open( scriptpath + agenttype + ".a" ).read()
```

The script then instantiates the mix of user agents. The code loops through the tuples in the agents list. The `exec` function calls a function definition in the doree.a, mira.a, and simon.a script files.

```
        time.sleep( startupdelay )
```

After each agent thread instantiation, the script uses the sleep method to give some time for the thread to bring up a client connection before starting the next one. Without this delay, a thread might time-out before it connects to the host.

```
record = 1
```

All the threads are instantiated so changing the value of the record global variable tells the agents to begin logging results to the `log` object.

```
start_time = Date().time
end_time = start_time + timetorun
while Date().time < end_time:
    time.sleep( 1 )
```

The master component then waits for the test period to complete. The `timetorun` value is set in the *properties.a* file.

```
record = 0
```

The test period is over. Setting the record global variable tells the running agents to stop logging results.

```
recording_time = Date().time - start_time
```

Finally, the `record_time` variable is set to the duration of this test in milli-seconds.

Cleanup

The run section of the master component instantiates the correct mix of test agent threads and then waits for the test period to complete. The cleanup section closes the `log` object and runs the *tally.a* script to determine the test results.

```
log.closeLogFile()

running = 0

duration = Date().time - start_time

exec open( scriptpath + "tally.a" ).read()
```

The master component is the conductor that stages, runs, and shuts down the test. The master component uses the global variables set in the *properties.a* script. The next section examines the *properties.a* script.

Property Files for Test Configuration

The stock trading system test has goals for reuse and maintenance. Using property files to store configuration and setup information makes it easier to achieve these goals. Property files are simple text files that you can modify with any text editor. Property files make it less important that the person operating the test knows the internals of the test agent code.

The *properties.a* file establishes these parameters:

- `agentcount`—The total number of concurrently running threads. The count value can be any positive integer. However, larger settings may impact the test results, as described below.
- `agents`—A list object that defines the user archetype for an agent thread to use and a percentage of the `agentcount` value that uses this archetype. For example, `agents=[doree,25]` tells the master component to use the Doree user archetype in 25% of all the agent threads created. If `agentcount` equals 60, then 15 of the agent threads will be Doree agent threads (25% of 60 agents total is 15 agents).
- `timetorun`—Duration of the test period. The value is in milliseconds (i.e.,1,000 milliseconds equals 1 second).

- `localname`—Name used to identify the machine running the test.
- `resultspath`—The path to the location of the log file.
- `startupdelay`—Amount of time to sleep between instantiating agent threads. This varies from machine to machine depending on the speed of the processors used.

The property file in the stock trading test suite is actually a runable TestMaker script. The master component runs the property file to establish variables used by the master and test agent components. The master uses the `exec` command to run the property file.

Implementing the Logging Component

The master component performs three functions: setting up the test, instantiating the agent threads, and recording the results to a log file. Next, we look at the way the master component creates log files. Every developer, QA analyst, or IT manager needs to consider what he or she will log because the old axiom "Garbage in garbage out" still holds true even in a test agent environment. In the stock trading test we sought to answer the following questions:

- Do I have the right number of servers?
- How does the system perform under increasing loads of use?
- Will enough activity eventually cripple the system?

With these questions in mind, it is fair to record the type of operation, a flag telling if the transaction completed successfully or failed, and the duration of each transaction. This way we can see the changes in throughput when the average transaction takes longer as more agent threads run concurrently. We can also determine a rate of TPS and a rate of errors-per-transaction (EPT).

The next consideration is how the agent threads log the results. Several possibilities exist, including:

- Keeping track of the statistics we wish to report as the test is in progress, but not recording the individual transactions. This method is fast and requires few resources (memory or disk space) but it does not allow us to go back later and re-evaluate the results.

- Logging each transaction to a `log` object that keeps the log entries in memory until the test is complete. At the end of the test, the in-memory transaction data is dumped to a log file. If a test runs for several days—as some tests will invariably do—then this approach is hugely memory resource intensive.
- Logging each transaction to a log service that consolidates log entries from multiple concurrently running agent threads on multiple machines into a single common log. This approach makes best sense when multiple machines are used to run the test agents.
- Logging each transaction to a file. This approach makes it easy to revisit the transaction data at later times.

There are other strategies for logging results data. For example, since multiple agent threads are running concurrently, each agent could log to its own file and all agent threads could log to a single file.

For the stock trading system, logging each transaction to a file for later analysis makes the most sense for simplicity and coding efficiency. To accomplish this we build the agent threads to log their transaction results to a single synchronized `log` object that saves the logged results to a single file. The TestMaker TOOL includes a simple logger object that provides such a synchronized logging function.

To create a new simple logger object, use the following code:

```
try:
    log = simplelogger.getInstance()
    log.setFileName( logtarget )
except java.lang.Exception, ex:
    print "Could not start logging. Stopping test."
    sys.exit()
```

Then to log a result, the agent thread uses this code:

```
log.log( result )
```

The result value used in the previous code is a simple string and can contain any string value.

Avoiding Test Scalability Problems

The master component reads from a property file to determine the type and number of agent threads to create. These threads leverage the host platform's ability to run threaded programs concurrently. If the underlying host platform has a single processor, then the operating system time-slices the threads to share the processor. Even with host platforms that have multiple processors, eventually there are more threads than processors and the operating system has to divide up the processing power to run the threads concurrently.

A concern to watch when using intelligent test agents to test a server for scalability is that the test agents themselves may have scalability problems. Eventually the number of concurrently running agent threads overwhelms the operating system's ability to share a processor. Each agent thread gets less and less time on the processor. From the agent's perspective it appears that the server is responding slowly when it may be that the agent's thread is not getting enough time to handle the server's response.

Perhaps the most significant indication that agent threads are running into scalability issues is to look at the minimum, maximum, and average time it takes agent threads to process transactions. Consider the test results in Table 11–3.

Table 11–3 Example Test Results

Agent type	Transaction	Duration (milliseconds)
Mira	Sold shares	3275
Mira	Sold shares	3890
Mira	Sold shares	2501
Mira	Sold shares	4289

These results were taken with 100 concurrently running agent threads. Looking at these results, the minimum time in milliseconds—there are 1,000 milliseconds in 1 second—it took to process one transaction was 2,501. The maximum time is 4,289. The average is 3,488.

Next we will increase the number of concurrently running agent threads to 500. Table 11–4 shows the results recorded after each transaction.

Table 11–4 Example Test Results Show the Test is Hitting a Wall

Agent type	Transaction	Duration (milliseconds)
Mira	Sold shares	37288
Mira	Sold shares	56701
Mira	Sold shares	18038
Mira	Sold shares	48500

Notice that the average time to complete a transaction jumped to 40 seconds! By adding five times the number of agent threads, the test agents turned in results that are 100 times slower. That is not good.

When configuring a new test, you must pay careful attention to the scalability of the test environment. Ignoring the test environment scalability risks turning in numbers that are meaningless or worse, misleading.

One test scalability technique I have found successful is to build into the test agent code the ability to run increasing numbers of test agents and plot the results on a simple chart, such as presented in Figure 11–7.

Charting the number of transactions processed against the number of concurrently running agent threads shows the test environment scalability. These tests run in networked environments where you can add additional test agent hosts to the network to create the desired levels of load. The chart helps you estimate how many test agent hosts are needed.

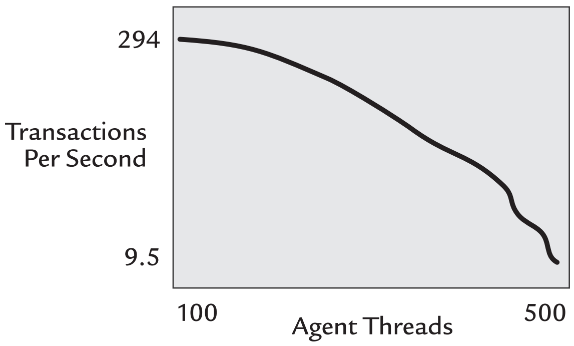


Figure 11–7 Charting the scalability of the test environment by showing the total number of agent threads in the X axis against the number of completed transactions per second in the Y axis. This aids in estimating the number of test agent hosts needed to run the test.

Another way to ease scalability problems in test agents is to incorporate the `yield` command into the agent code. `yield` tells the operating system that the agent thread is at a point where yielding control to another thread is acceptable. `yield` is best used before and after a large calculation needs to be made, or during a loop that exits when an external process is finished. For example, when an agent is polling for a message from an email server, the agent thread could yield between polling commands. This is reminiscent of my kindergarten days when the teacher instructed all the kids in a sandbox to share.

A First Look at the Results

While the next chapter shows how the *tally.a* script is constructed and how to analyze the logs to determine actionable knowledge, the temptation to immediately look at the test agent results is usually very great. So, let us take a peek at those results before we move into the next chapter.

The stock trading test constructed in this chapter provides a configurable way to test for scalability, performance, and functionality on the system level. The default settings run the user archetypes concurrently. Right away the test agent results show system capacity, successful operations, and problems.

What we tested	What we found	Conclusion
Do I have the right number of servers?	The average transaction was less than five seconds. Transactions for Mira took on average 20% longer than the others. The longest response took more than 35 seconds.	The average transaction time is acceptable. However, when looking at transactions with the longest response times, we see why the real Mira thinks the system is running slow. In her case, it really is running slow. We need to look into why Mira's transactions are performing so slowly.
How does the system perform under increasing loads of use?	As additional test agents were used, Simon's transactions slowed by 300% when we reached the maximum load. However, at maximum load Mira's transactions were only 20% slower.	Simon's transactions uniquely communicate to the external MarketFacts service using SOAP calls. It appears that the MarketFacts service cannot handle the additional load.

Will enough activity eventually cripple the system?	Average transaction times degraded by more than 150% when 25 additional user agents were added to the test. Doree's transactions slowed by 300% when we reached the maximum load.	While the system is adequate to handle the existing team, the system is not ready for the company to hire additional traders.

I strongly caution you to be careful with the first look at the performance results. I have been tempted to jot off an excited email message to the project team announcing the test results. I caution you to avoid my past error!

The problem with a first look at the performance results is that it is only a first look. Usually much analysis and organizational effort is needed to ensure the results are accurate and meaningful.

Please, please, please read Chapter 12 before you sit back satisfied with the first look at performance results.

Summary

This chapter took the techniques presented in earlier chapters to command services over a variety of protocols (HTTP, HTTPS, SOAP, XML-RPC) to build a test modeled after a set of user archetypes. We went through all this trouble—defining user archetypes, modeling user behavior, and understanding user goals—so that the test would get close to emulating a real-world production environment. The extra effort makes certain the priority is to test the system against prototypical users' goals, rather than just test that the individual functions work.

The test handled configuration, test agent instantiation, and results recording. The next chapter shows you how to turn the recorded results into actionable knowledge.

12

Turning Test Agent Results into Actionable Knowledge

The stock trading information system in Chapter 11 presents a methodology, infrastructure, software design, and protocol design to implement a Web-enabled application with great scalability, reliability, and functionality. We designed user archetypes, wrote multiprotocol intelligent test agents, and made requests to an application host. First, we checked for the correct functional results, then we checked the host's ability to serve increasing numbers of concurrent users. All of this activity provides a near-production experience from which we can uncover scalability problems, concurrency problems, and reliability problems. It also usually generates a huge amount of logged data.

Looking into the logged data allows us to see many immediate problems with the Web-enabled application under test. The log file is one of many places you can observe problems and find places to optimize the Web-enabled application. This chapter shows how to understand and analyze a Web-enabled application while the test is running and how to analyze the results data after the test is finished. With the method presented in this chapter, you will be able to demonstrate the system's ability to achieve scalability, reliability, and functionality.

Chapter 11 took the techniques presented in earlier chapters to command services over a variety of protocols (HTTP, HTTPS, SOAP, XML-RPC) and build a test modeled after a set of user archetypes. It presented the test goals, user archetypes, and test agents for an example stock trading firm. The master component of the test handled configuration, test agent thread creation, and recorded the results to a special log file. This chapter explains how to turn the recorded results into actionable knowledge.

What to Expect from Results Analysis

Chapter 11 ended with a strong word of caution. You may be tempted to conduct a cursory review of test results for actionable knowledge. In this regard Alexander Pope had it right when he wrote: "A little learning is a dangerous thing; Drink deep, or taste not the Pierian spring." Thoroughly analyzing test results produces actionable knowledge, whereas looking only at the surface of the test result data can lead to terrible problems for yourself, your company, and your project. So, before showing how to analyze the test result data generated from the intelligent test agents in the previous chapter, this section presents what we can reasonably expect to uncover from conducting a test.

Results data provides actionable knowledge, but the meaning may be contingent on your role in the software process. Consider the following tests and how the actionable knowledge changes depending on who is running the test. Table 12–1 describes this in detail.

Table 12–1 Actionable Knowledge Changes Depending on Who is Running the Test

Activity	Test	Who	Actionable knowledge
A software developer writes a new function	Functional test	Developer	Determines that the function works and the new module is ready for testing.
Delivery of new software build	Scalability and concurrency test	QA technician	Identifies optimization possibilities to improve performance and reduce resource needs (CPU, disk, memory, database).
Production servers upgraded	Rollout test	IT manager	Determines when the datacenter infrastructure is capable of serving forecasted user levels.

In each case, the same intelligent test agents may stage a test but the results log is analyzed to find different actionable knowledge. For example, when a QA analyst looks at the log file on a server undergoing a scalability and concurrency test, the analyst will be looking for log entries that indicate when a thread becomes deadlocked because it is waiting on resources from another thread. The developer looking at the same results log would be satisfied that the module under test functioned. Therefore, a starting point in analyzing results is to understand the goals of the test and see how the goals can be translated to results.

Following are a few test goals and how the goals may be translated to actionable results.

Goal: Our New Web Site Needs to Handle Peak Loads of 50 Concurrent Users

Imagine a company Web site redesign that added several custom functions. Each function is driven by a Java servlet. The goal identifies the forecasted total number of concurrent users. The definition for *concurrency* is covered later in this chapter. For the moment concurrency means the state where two or more people request a function at the same time.

One technique to translate the goal into an actionable result is to look at the goal in reverse. For example, how would we know when the system is not able to handle 50 concurrent users? Imagine running multiple copies of an intelligent test agent concurrently for multiple periods of time. Each test period increases the number of concurrently running agents. As the test agents run, system resources (CPU time, disk space, memory) are used and the overall performance of the Web-enabled application slows. The logged results will show that the total number of transactions decreases as more concurrent agents run.

Charting the results enables us to set criteria for acceptable performance under peak loads. For example, at 100 concurrent test agents the total number of transactions completed might be three times smaller than when 50 concurrent test agents are run. Charting the transactions completed under an increasing number of concurrent test agents enables us to pick a number between 50 and 100 concurrent test agents where system throughput is still acceptable.

Goal: The Web Site Registration Page Needs to Work Flawlessly

Imagine a company that promotes a new fiction book. The company Web site provides a Web page for prospective customers to register to receive announcements when the book is published. A simple HTML form enables prospective customers to enter their contact information, including their email address. A Microsoft ASP.NET object serves the HTML form. When a user posts his or her contact information, the ASP.NET object needs to record the information to a database and redirect the user to a download page.

This reminds me of a time I waited for a phone call from a woman I invited out to dinner on a date. The longer I waited, the more I thought the phone might not be working. To my chagrin, I lift the phone receiver and find that the phone was indeed working. Doing so, of course, prevented her call from getting through to me. The analog to this is testing an HTML form. Until you actually click the submit button in a browser interface, you don't really know that the server is working. Yet, clicking the button causes the server to do actual work for you that takes resources away from real users.

One technique to translate the goal into an actionable result is to understand the duration of the goal. Consider that the only way to know that the HTML form and ASP.NET object are working flawlessly is use them. And each time they are used and perform correctly we have met the goal. So how long do you keep testing to achieve the goal of "flawless performance"? Understanding the goal of the test can be translated into a ratio of successes to failures.

The service achieves the goal when the ratio of successful tests of the HTML form and the ASP.NET object exceed by a set among the tests that failed. For example, over a period of 24 hours the goal is achieved if the ratio of successful tests to tests with failures always exceeds 95%. Searching the logged results for the ratio is fairly straightforward. Alrighty then!

Goal: Customer Requests for Month-End Reports Must Not Slow Down the Order-Entry Service

A common system architecture practice puts a load-balanced group of application servers in front of a single database server. Imagine the application server providing two types of functions: one function uses many database queries to produce a month-end sales report to salespeople and the second uses database insert commands to enter new orders into the database. In a Web environment both types of functions may be used at the same time.

One technique to translate the goal into an actionable result is to look at the nature of the goal. When the goal speaks of multiple concurrent activities, then an actionable result provides feedback to tune the application. The tuning shows system performance when the ratio of activity types changes.

In this example, the goal betrays that the system slows down toward the end of the month as customers increasingly request database query-intensive reports. In this case the goal can be translated into actionable results by using a combination of two test agents: one agent requests month-end reports and the second places orders. Testing the system with 100 total agents and a changing mix of test agents shows a ratio of overall system performance to the mix of agents. For example, with 60 agents requesting month-end reports and 40 agents placing orders, the system performed twice as fast as with 80 agents requesting month-end reports and 20 agents placing orders. The changing mix of agent types and its impact of overall performance makes it possible to take action by optimizing the database and improving computing capacity with more equipment.

Goal Summary

The examples and goals presented here are meant to show you a way to think through the goals to determine a course of action to get actionable knowledge from test results. Many times simple statistics from logged results are presented. While these statistics might look pretty, the actionable knowledge from the test results is the true goal you are after.

The Big Five Problem Patterns

Looking through raw logged results data often gives a feeling of staring up at the stars on a cold, clear winter night. The longer one looks into the stars, the more patterns emerge. In testing Web-enabled applications for scalability, performance, and reliability, five patterns emerge to identify problems and point to solutions.

Resource Problems

While there may be new software development techniques on the way, today's Web-enabled application software is built on a "just-in-time" architecture. An application responds to requests the moment it receives the request. Web-enabled applications written to run on a host typically wait until a given resource (CPU bandwidth, disk space, memory, network bandwidth) becomes available. This is a major cause for application latency.

At any moment the hosting server must be able to provide the needed resource to the application, including resources from Web service hosts. The log results record the latency that occurs while the host and Web services provide the needed resources to the application.

The host running a Web-enabled application provides resources in the form of memory, disk space, processor bandwidth, and network connectivity. Two strategies emerged over the years to understand resource allocation in server-side applications. Remote agent programs running separately monitor resources and provide you with a remote programmatic interface for retrieving the resource information. The leader in this space is the System Network Monitor Protocol (SNMP) standard. Java developers have access to SNMP agent data through the Java Management Extensions (JMX) library. Details on both are found at http://java.sun.com/jmx. Management consoles, such as HP OpenView, provide a dashboard that collects, collates, and displays SNMP agent data.

The second strategy for understanding resource allocation is to build resource monitoring into the Web-enabled application. This is accomplished by providing the software developer with a set of APIs that provide live data about the resources available on the host machine and those used by the application. The developer writes the Web-enabled application to write the resource usage to the results log as each transaction is being handled.

As we found in the test agent presented in Chapter 11, we chose the later strategy for understanding resource allocation. The test agents logged the transaction data to a log file. For example, the agents could also use the built-in resource reporting APIs to save current memory size, time from the last Java Virtual Machine garbage collection, and amount of free disk space. Most test tools, including those introduced in Chapter 5, have methods available to scripts to learn about current resource usage and availability.

Writing code in a test agent to check resource allocation is the better strategy because the developer writing the test agent knows what resource information is important. Generically recording system resources using a management console and SNMP agents has a tendency to produce extraneous log results.

While any unexpected latency reported in a results log might appear to be caused by a lack of resources, latency is oftentimes caused by one of the other four big types of problems.

Concurrency Problems

The days when a computer's video display froze for a few seconds when the floppy disk drive started up ended when multiple dedicated processors were added to the motherboard. Even on a single processor-equipped desktop or server machine, the motherboard contains separate processors for handling disk, sound, video, and network operations. The modern computer is a multitasking machine by design. Event-driven applications, on the server or client side, are built to handle multiple concurrent tasks.

Concurrency is a measurement taken when more than one user operates a Web-enabled application. One can say a Web-enabled application's concurrency is good when the Web service can handle multiple users' operating functions and making requests at the same time with little speed degradation while handling each user's operations.

Concurrency measurements are recorded in two ways. When an application runs on multiple load-balanced machines, a simple analysis of the combined results log shows the concurrency of the applications running on the machines behind the load balancer as a unit. Second, as each machine handles concurrent requests for the application's functions, the results log shows how the operating system and application handle threads and context switches.

In both ways, I measure concurrency by determining when a transaction's start time is after a second transaction's start time, but before the second transaction's end time. A test agent script can parse through the log results and tally the number of concurrent transactions.

In today's world of datacenters where so few people understand what is impacting system performance but almost everyone is a user of Web-enabled applications, it has become popular opinion that "concurrency" is the cause of bad things. Contrary to popular opinion, concurrency is *not* a bad thing. Few applications require all of the host's resources all the time. Unfortunately for server applications, concurrency has gotten a bad reputation as the root of increasingly slow application performance. In reality, measuring concurrency is a great way to measure how efficiently the system shares its resources.

The typical pattern that identifies concurrency problems is to measure concurrency over a period of time. During this time period, the test increases the number of concurrently running transactions. For an indicator of a concurrency problem, look at the ratio of TPS at the start and end of the test. If you observe that the number of transactions completed decreases as the number of concurrent transactions increases, then you can suspect a concurrency problem.

Concurrency problems are usually caused when bottlenecks are built into a system. Multiple requests stack up waiting for a single synchronized method to complete its function for the previously received requests. An analysis of the results log also shows a ratio of transaction times for transactions that are not running concurrently to those that are running concurrently.

Understanding the pattern for concurrency problems is a significant asset when solving scalability and performance problems. However, sometimes concurrency problems mask a component problem.

Component Problems

The problem patterns identified so far appear because they reoccur in the results log. The more activity the system processes, the more the problem occurs. On the other hand, component problem patterns are nonrecurring or occur seldom enough that there is no repeated pattern that becomes obvious from analyzing results logs. Additionally, component problems appear in a results log as errors, whereas the majority of results log entries are successful transactions. When the component fails, it fails rarely. For component problems we need a different strategy for results log analysis.

The top priority for developers, QA analysts, and IT managers when solving component problems is to determine which component fails and what scenario of actions, use, and load contributes to the failure. For example, consider a private collaborative extranet service that offers a large enterprise sales organization the ability to share documents, chat, and participate in bulletin-board-style threaded messages. The extranet service, hosted by Inclusion Technologies, uses an XML single-sign-on mechanism described on IBM developerWorks at http://www-106.ibm.com/developerworks/webservices/library/ws-single/. After signing in, a salesperson reads briefing materials and posts questions to a group discussion list. The salespeople may optionally subscribe to participate in the group discussion through their email client. As messages are posted to the discussion list, the Web users see the messages in a threaded list and the email subscribers receive a copy of the posted message in their email account. Replies to the email messages are posted back to the discussion group.

Looking through the logs of the collaborative extranet system showed that approximately every three days a salesperson was not receiving a day's worth of email messages. The problem was intermittent and appeared to resolve on its own, only to fail later.

The solution was to build and run an intelligent test agent modeled after a salesperson's behavior. The agent signs in, posts messages, receives, and replies to email messages. When the system fails, the agent marks the time and steps that caused the problem. Identifying the scenario that causes the problem shows the software development team where a thread is not handling exceptions thrown by the email server. The next day when the user signs in, a new thread is instantiated to replace the hung one and the email delivery problem is magically solved. Understanding what part of the Web-enabled application failed leads the developers to build and deploy a fix to the code.

The key to solving component problems is finding the malfunctioning component. Test agents help to hit a component in just the right way and observe the failure.

Contention Problems

Competition between Web-enabled applications, and on a lesser scale the threads in a single application, lead to the system making decisions on which thread gets the focus and when. Contention problems happen when one type of thread predominantly gets the focus. For example, in the Stock Trading Company example in Chapter 11, the system responded to many requests concurrently. Imagine what would happen when the system gave preference to database queries. More requests from workers doing stock market research would cause the stock trader's requests to slow down.

Running multiple concurrent intelligent test agents against an information system provides a unique view of the contention problems in the system. The test agents can change the mix of concurrently running test agent types and analyze the impact on performance and scalability. For example, consider the test results in Table 12–2.

Table 12–2 Test Results from Multiple Concurrent Test Agents

Test agent	Quantity running concurrently	Average transaction time
Mira	40	16 seconds
Simon	30	22 seconds
Doree	30	35 seconds

Now, imagine we adjust the mix of concurrently running test agents so that more Simon agents run. Table 12–3 shows the test results.

Table 12–3 Notice the Change from Changing the Test Agent Mix

Test agent	Quantity running concurrently	Average transaction time
Mira	20	50 seconds
Simon	60	20 seconds
Doree	20	35 seconds

Note that we doubled the number of Simon agents from the previous test and the average transaction time for Simon decreased. At the same time we ran half the number of Mira agents, but the average transaction for Mira slowed to 50 seconds. While many problems might lead to these performance inconsistencies, experience tells us to look for contention problems: Look at the system memory usage, look at the database resource logs, and look at the application server settings.

Crash Recovery Problems

When a desktop or laptop computer crashes, our first urge is to restart the computer to solve any lingering problems brought on by the crash. Servers are meant to run for a long time and handle even exceptional cases like application crashes. So restarting the server is usually not an option. Instead, we must build high-quality crash handling code into Web-enabled applications.

Intelligent test agents have a place to play in detecting faulty crash recovery code. Test agents are in a unique position to be able to cause crashes and drive the system to handle additional requests concurrently. For example, consider a test scenario where the Simon test agent from Chapter 11 makes dozens of requests for research reports. The system makes dozens of queries to the database. If one or more queries throws an exception, the server must handle the exception and release any used resources. The test agent is in a unique position to learn what happens if the exception handling code does not release the thread's memory, open file handles, and database connections.

In summary of the Big Five Problem Patterns, we found general patterns emerge when analyzing results logs. The patterns provide a way to detect problems and produce actionable knowledge that software developers, QA analysts, and IT managers can use to solve problems in Web-enabled applications.

Key Factors in Results Analysis

The previous section showed the top patterns to identify actionable knowledge in results logs. While the patterns show what to look for, the next section shows how to measure what you find. The measurements tend to fall into the following four groups:

- *Concurrency* is a measurement taken when more than one user operates a Web-enabled application. A Web service's concurrency is good when the Web service can handle large numbers of users using functions and making requests at the same time. Many times concurrency problems are solved with load balancing equipment.
- *Latency* is a measurement taken of the time it takes a Web-enabled application to finish processing a request. Latency comes in many forms, for example, the latency of the Internet network to move the bits from a browser to server, and software latency of the Web service to finish processing the request.
- *Availability* is a measurement of the time a Web service is available to take a request. Many "high-availability" computer industry software publishers and hardware manufacturers claim 99.9999% availability. As an example of availability, imagine a Web service running on a server that requires two hours of downtime for maintenance each week. The formula to calculate availability is: 1 – (downtime hours / total hours). Since there are 168 total hours each week, a weekly 2-hour downtime results in 98.8095% availability.
- *Performance* is the measurement of the time between failures. This is a simple average of the time between failures. For example, an application that threw errors at 10:30 am, 11:00 am, and 11:30 am has a performance measurement of 30 minutes.

These four types of metrics enable us to speak intelligently between software developers, QA analysts, and IT managers to quantify good system performance. The metrics become part of an overall report on scalability, performance, and reliability. Experience shows that the final report usually focuses on two areas: throughput and reliability. Throughput is measured in terms of transactions per second and answers the question, "Did I buy enough servers?" Reliability is measured by a SPI and answers the question, "Does this system allow users to achieve their goals?"

Measurements of concurrency, latency, availability, and performance are normally calculated in terms of system throughput. This measurement shows how many TPS the information system can handle.

TPS is meant to be a simple measurement. The TPS measurement counts the number of successful transactions and divides by the total number of seconds the test took to complete. Transactions are defined as the total time it takes for a group of related requests to complete one business function. For example, in the Stock Trading Company example presented in Chapter 11, the Mira agent signs in to the system, requests stock quotes, and places an order. These steps combined are considered a transaction.

This book espouses a user goal–oriented testing methodology to look at testing from a user perspective. It should be no surprise then that this book focuses on TPS measurements from the client side of a Web-enabled application. That does not mean TPS is measured exclusively from the client side. When debugging subsystem scalability problems, it may be appropriate to measure TPS at the database subsystem by having a test agent communicate directly to the backend database. Comparing client-side TPS to the subsystem TPS would show a source of latency in the system.

While TPS reports on throughput, the next method addresses user goals. Quantifying user achievement has been a difficult task made worse with the invention of Web-enabled applications. What is needed is a new methodology for taking criteria for acceptable Web-enabled application scalability and performance and describing the measured results in terms of a SPI. Chapter 2 first introduced SPI. It is a metric with many uses, including the following:

- Encourages software developers to optimize code for better user experiences.
- Helps IT managers to design and deploy the right size datacenter.
- Provides management metrics to judge user satisfaction.

- Aids QA analysts to learn if new software has regressed to include previously solved problems.

The TPS and SPI methods for measuring concurrency, latency, availability, and performance provide a set of metrics that software developers, QA analysts, and IT managers use to manage system reliability. Next, we see how these measurements may provide misleading results and how to avoid making such mistakes.

Scenarios Where Results Data Misleads

Over the years of pouring over results logs and implementing scalability and performance tests, I discovered the following scenarios—better to call them rabbit holes and wild goose chases—where the results data looked correct, but the conclusions were incorrect and misleading.

The Node Problem

The user goal-oriented test methodology presented in this book uses intelligent test agents to drive a Web-enabled application. You might ask, "Where do I run the test agents?" TestMaker provides an easy environment to run test agents on the local machine. Many times a test needs to be mounted that exceeds the local machine's abilities to run the needed number of concurrently running test agents. This book recommends the following two strategies:

1. Configure multiple machines to run TestMaker. Use a remote control utility—for example, ssh, the secure shell terminal on Linux—to control TestMaker remotely. TestMaker comes with shell scripts for Linux and Windows systems to run test agents from a command-line shell. After the test agents complete their test, then manually copy the results logs back to a single machine for analysis.
2. Use PushToTest TestNetwork, a commercial product that compliments TestMaker by providing the following capabilities:
    - Run agents in greater scale than on a single machine. For example, on a single 2 GHz Pentium system, TestMaker runs up to 500 concurrent test agents. With TestNetwork, running 10,000 or more concurrent test agents is possible.

- Run test agents on multiple remote machines. TestNetwork turns remote systems into TestNodes that remotely run test agents and report the results back to a central TestMaker console.
- Keep test agents running for a long duration. TestNetwork's console/TestNode architecture turns test agents into mini-servers that are able to handle operations autonomously.

In my experience, variations in the results data can be introduced when the operating environment under which the test nodes—each machine is a node—is poor. In one Web-enabled application test, I noticed that the performance results from one test node was 20% slower overall than the other test nodes. It turned out that the test node had a backup utility enabled that caused all applications—the test agents included—to run slower while the backup was being completed. The solution to this problem was to dedicate a test node to only run test agents. To avoid problems like this, PushToTest sells a completely configured rack-mounted TestNetwork test node appliance. For details, see the PushToTest Web site.

The Hidden Error

An experience at 2Wire, the leading DSL deployment technology provider to the world's largest ISPs, showed that test agents need to look deeply into system responses. Chapter 15 presents a case study of a scalability test for 2Wire. For example, 2Wire hired PushToTest to run a scalability test against 2Wire's datacenter to determine their readiness to handle customer requests. The 2Wire devices use a modified XML-RPC protocol to communicate to the datacenter. While it looked like the datacenter host was providing XML-RPC responses, the majority of responses contained errors encoded in the XML response.

The experience shows that test agents might not just accept a response as a successful transaction. The test agents need to validate the response. Errors may appear in the following three levels of a response:

- Transport-level errors. Typical of this level of error are connection problems with the host ("connection unavailable," "http fault"), wrong payload size exceptions, and unsupported protocol errors. Another problem may be a response that simply says, "The server has no more available connections."

- SOAP response exception. The SOAP header contains an exception. For example, the response says the first element of the request does not conform to the expected input where a long value is received when an integer value is expected.
- Message body exception. The body of the message indicates an exception. For example, a response with a response element of the message body containing: "MS SQL COM bridge exception: 82101."

At least HTTP and SOAP protocols provide for error reporting. The hidden error is even worse in testing email protocols (SMTP, POP3, IMAP) where no standards exist for error reporting. For example, when receiving an email message from a POP3 or IMAP host, how does a test agent know if the message is a "bounce" from a mail host because the recipient is unknown to the system? POP3 and IMAP do not define an error protocol so it is up to the test agent to look into the body of the message to try to detect a "bounce" status.

Dileep's Dilemma

Elsevier Science was very interested to learn SOAP scalability and performance metrics of SOAP-based Web services on several application servers, including IBM WebSphere, SunONE Application Server, and BEA WebLogic Server. The SunONE team at Sun Microsystems was nice enough to offer the use of their lab, equipment, and engineers to conduct a study.

Dileep Kumar is a Sun software engineer who helped conduct the tests. Dileep was well trained at using Segue Silk, a commercial Web application test tool, to simulate a SOAP request to a Web service running on a multiple-CPU SPARC server with the SunONE application server.

He wrote a Silk script that made a pseudo-SOAP request to the server. The Silk script did not actually use a SOAP stack; instead, it formed and issued an HTTP `Post` command to the running Web service. He measured the time it took to issue the HTTP request and receive a response. To his surprise the transaction time was 40% faster than the results found with TestMaker.

He missed that the TestMaker result metrics included the time it took to marshal the request through a SOAP stack and XML parser on the client side. This experience is a good lesson to teach us that when examining any results, first ensure you are comparing similar items.

Diminishing Returns

Many times, results analysis often turns into an exercise of diminishing returns. Each effort to look deeper into the logged results data yields smaller and smaller benefits to system reliability and scalability. The best test analysts can easily get caught up in an effort to continue sorting, filtering, and pivoting the results data because they suspect additional scalability and reliability knowledge can be found if they look just a little longer. To avoid this problem, I recommend you understand the goals of the test first, then analyze the data.

Back to the Stock Trading Example

With all of this talk about turning result logs into actionable knowledge you may be wondering about the Stock Trading Company example that started in Chapter 11. Will Mira, Simon, and Doree find happiness after all? Yes, it turns out. But, not until we tally the results log to learn the system capacity as measured in TPS.

Implementing the Tally Agent

In Chapter 11, the *master.a* agent script mounts a test of the Stock Trading Company example by configuring several test agents, running the agents, and logging the results to a file. The *master.a* agent script's final operation is to run the *tally.a* script. *tally.a* parses through the results log and determines the TPS metric for the test. This section shows how the tally component is constructed.

First, we examine the *tally.a* script in its entirety. Then, we will provide a detailed script explanation. All of the code presented in this book is also available for download at http://www.pushtotest.com/ptt/thebook.html.

```
# Script name: Tally.a
# Author: fcohen@pushtotest.com

# exec open( scriptpath + "Properties.a" ).read()

# Set-up variables

tallyman = "Come Mister tally man; tally me bananas."
logtarget = resultspath + "results_" + localname + ".txt"

# First pass through the data
```

```
print
print "Tally the results:"
print "Find the minimum, maximum, average transaction time,
error index"
print

mark1 = 1
response_total_time = 0
response_count = 0
resp_min = 0
resp_max = 0
error_count = 0

start_index = 0
end_index = 0

try:
    print "Analyzing data in", logtarget
    rfile = open( logtarget, "r" )
except IOError, e:
    print "Error while opening the file:"
    print e
    print
    sys.exit()

result = rfile.readline()

while result != "":

    # The file is in comma delimited format
    params = result.split(",")

    response_time = int( params[0] )
    status = params[1]
    time_index = params[2]

    if status == "ok":
        response_total_time += response_time
        response_count += 1

        if mark1:
            mark1 = 0
            resp_min = response_time
            resp_max = response_time

            start_index = time_index
            end_index = time_index
        if response_time > resp_max:
            resp_max = response_time

        if response_time < resp_min:
            resp_min = response_time

        if time_index > end_index:
            end_index = time_index

        if time_index < start_index:
            start_index = time_index

    else:
        error_count += 1

    result = rfile.readline()

rfile.close()

print
print "Agents: ", agentcount
print
print "Minimum transaction time: %d" % resp_min
print "Maximum transaction time: %d" % resp_max
print "Average transaction time: %.2f" % ( float(
response_total_time ) / float( response_count ) )
print "Error count:", error_count

# Calculates the duration in milliseconds that the test took
to run
test_duration = long( end_index ) - long( start_index )

print
print "Total test time: %d" % test_duration
print "Number of completed transactions: %d" % (
response_count )
print "Transactions Per Second: %.2f" % ( float(
response_count )\
/ ( test_duration / 1000 ) )
print
print
```

The first part of the tally component identifies the objects that are used in the script and initialize variables. Following is the same code with a detailed explanation.

```
tallyman = "Come Mister tally man; tally me bananas."
logtarget = resultspath + "results_" + localname + ".txt"
mark1 = 1
response_total_time = 0
response_count = 0
resp_min = 0
resp_max = 0
error_count = 0

start_index = 0
end_index = 0
```

These variables assist the script to find the minimum, maximum, and average transaction time and an error index.

```
try:
    print "Analyzing data in", logtarget
    rfile = open( logtarget, "r" )
except IOError, e:
    print "Error while opening the file:"
    print e
    print
    sys.exit()
```

The script opens the log file, which was created by the master component and populated with data from the individual test agent threads.

```
result = rfile.readline()
while result != "":
```

The script parses the contents of the results log file line-by-line. The `readline()` method returns an empty value when the end of the file is reached.

```
    params = result.split(",")

    response_time = int( params[0] )
    status = params[1]
    time_index = params[2]
```

The data in the results log file is delimited by commas. The handy `split()` method finds each value and returns a list of the values in the `params` variable.

```
    if status == "ok":
```

The transactions per second metric in this analysis counts only the successful transactions.

```
        response_total_time += response_time
        response_count += 1
```

The `response_total_time` and `response_count` values keep track of the responses to calculate the average response time later in the script.

```
        if mark1:
            mark1 = 0
            resp_min = response_time
            resp_max = response_time

            start_index = time_index
            end_index = time_index
```

The first result in the log becomes both the minimum and maximum response time.

```
        if response_time > resp_max:
            resp_max = response_time

        if response_time < resp_min:

        if time_index > end_index:
            end_index = time_index

        if time_index < start_index:
            start_index = time_index
```

As the script parses through each line of the results log, the minimum and maximum response time is stored for later use. The script also keeps track of the actual start and end time.

```
    else:
        error_count += 1
```

Although the TPS value only includes successful transactions, the tally script also reports the count of transactions ending in errors.

```
print
print "Agents: ", agentcount
print
print "Minimum transaction time: %d" % resp_min
print "Maximum transaction time: %d" % resp_max
print "Average transaction time: %.2f" % ( float(
response_total_time ) / float( response_count ) )
print "Error count:", error_count

# Calculates the duration in milliseconds that the test took
to run
test_duration = long( end_index ) - long( start_index )

print
print "Total test time: %d" % test_duration
print "Number of completed transactions: %d" % (
response_count )
print "Transactions Per Second: %.2f" % ( float(
response_count ) / ( test_duration / 1000 ) )
```

As we see, *tally.a* is a simple script to calculate the TPS metric from the results log. The tally script displays its results to the TestMaker output window.

Summary

This chapter concluded the work we started in the previous chapter. In Chapter 11 we created intelligent test agents to emulate prototypical users. The test agents drive near production environment loads on a Web service and consequently the system generated a boatload full of raw logged data. We wrote agent code to analyze the results data into actionable data. The TPS report shows us what to expect from the system when it is in a production environment.

13
Concurrency and Scalability in a High-Volume Datacenter

2Wire is a manufacturer of innovative DSL systems. The 2Wire HomePortal residential gateway and OfficePortal broadband router are the telecom industry's first truly intelligent, customer-installable, multiservice devices. 2Wire uses a modified XML-RPC protocol to enable the HomePortal gateway device to communicate with a datacenter to receive its programming, updates, and content. In this environment, 2Wire needs to answer an important question:

How do we service and scale CMS efficiently for millions of HomePortals?

This chapter describes how 2Wire went about answering this question. The chapter begins by describing the Web-enabled application environment of the 2Wire Component Management System (CMS), the goals for a concurrency and datacenter readiness test, the test methodology used, and how the results are compiled and analyzed. The concepts and methodologies presented earlier in this book are put to the test in this chapter.

Introduction

2Wire (http://www.2wire.com) provides technology to leading Internet service providers, including SBC and Verizon. The 2Wire HomePortal gateway devices provide self-configuration of DSL network parameters in an end-customer's home or business by communicating with a 2Wire CMS datacenter. I have one of these beautiful HomePortal boxes in my garage. My local phone company, SBC, charged all of $50 for the 2Wire HomePortal gateway device. Inside the device is a DSL modem, NAT and DHCP router, firewall, and an intelligent processor that offers value-added services, including a content filtering service. My experience configuring this kind of device in the past was not a fun one. I spent many hours changing configuration settings to get the device up and running. When I received the 2Wire HomePortal device, I plugged it in, launched its configuration utility, typed in a simple key code—a series of 16 digits—and the device configured itself. Nice!

2Wire manufactures this innovative DSL system. The 2Wire system operates over a Web-enabled infrastructure—TCP/IP routed networks, HTTP protocols, and Flapjacks-style load balanced application servers—using a modified version of the XML-RPC protocol. The HomePortal device includes a customized processor that implements the intelligent business workflow behavior to automatically configure, update, and otherwise manage the device functions. While 2Wire operates a central datacenter at the current time, their plan is to give the blueprints to their customers, the major telecom companies, to build their own datacenters.

2Wire learned about TestMaker through the open-source community and contacted PushToTest (http://www.pushtotest.com) to see if TestMaker was appropriate for their testing needs. 2Wire needed to prove to its customers that it was ready for larger deployments. PushToTest tested the 2Wire datacenter for readiness, tested the 2Wire CMS system for suspected concurrency problems, and delivered a customized test environment based on TestMaker.

The 2Wire Component Management System

Everyday ISPs seek new ways to automate their service. Service automation is the key to increasing revenue, reducing support costs, and opening new value-added opportunities to serve customer needs. 2Wire is the leading technology company to provide ISPs with an end-to-end solution to automate deployment and support of business and home Internet gateway solutions.

The 2Wire CMS solution combines a hosted datacenter with intelligent customer premises equipment (CPE) gateway devices. As a result the 2Wire system is the most advanced solution to automate a business and home user's access to the Internet.

For every ISP, automation is no longer an option. They require cost-effective solutions to automate basic configuration and management tasks, including DHCP and NAT routing configuration and DNS service configuration and security configuration. Plus, customer understanding of Internet network configuration and management has increased expectations of ease of use and reduced their willingness to deal with maintenance issues, especially ones that require phone calls to technical support.

2Wire delivers the following business benefits to ISPs.

- **Decreased costs**—CMS solutions provide gateway portal devices that are self-configuring from information received from a CMS datacenter. Customers enter a single keycode and the system handles all configurations of NAT, DHCP, and DNS settings. Configuration is fully automatic.
- **Increased revenue**—The CMS gateway portal devices are dynamically programmed to offer value-added services. For example, a value-added Content Screening Service provides protection to keep inappropriate material from being presented to computer users. CMS is a fully extensible platform to deliver value-added solutions.
- **Increased customer satisfaction**—The CMS solution presents friendly, easy-to-use Web-based interfaces for end-customers to configure and change options. These supplement existing telephone support options that typically feature frustrating and long wait times. CMS solutions are user friendly.

To accomplish these goals, 2Wire engineered the CMS system and integrated proven Internet technologies, established datacenter architecture, and technology that is available from multiple sources. Figure 13–1 shows the CMS solution architecture.

2Wire's CMS design implements a reliable, scalable, and well-performing system. The CMS infrastructure has friendly end-user interfaces and powerful administrative functions for system operators. For example, CMS can *push* an upgrade to a selection of HomePortal devices automatically or to the entire population of devices.

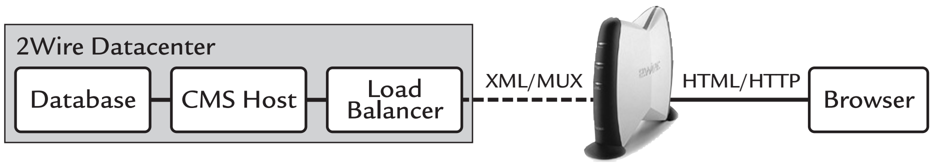


Figure 13–1 The 2Wire CMS system provides a Web browser interface to the configuration options in the 2Wire Home Portal device. The device then uses a modified XML/RPC mechanism to make command and information requests to the datacenter.

CMS provides a pluggable interface to add new value-added services over time. The design intends to leverage a Web-enabled infrastructure today that may be used to provide new value-added features in the future.

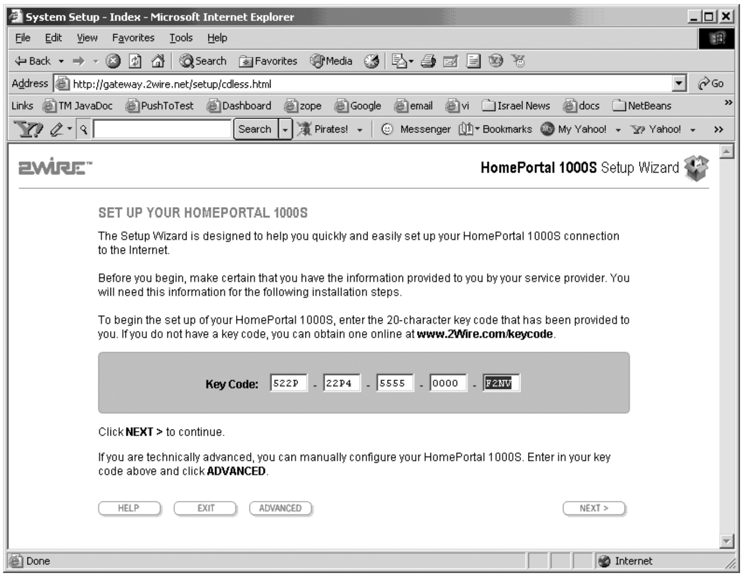
CMS systems enable 2Wire customers to handle large deployments. Understanding how CMS solutions perform in large deployments is the key to 2Wire's strategy to provide scalable solutions to ISPs. 2Wire's focus on reliability, scalability, and extensibility demands a deep understanding of the system.

Understanding What Happens over the Wire

To get an understanding of the 2Wire system architecture, I will start from the HomePortal gateway device configuration. Figure 13–2 shows a user interface the gateway device serves to the user's browser. The user enters a 20-digit keycode provided by the ISP. The gateway device uses the keycode to identify the URL of the datacenter.

Receiving a new keycode, the gateway device communicates with the 2Wire datacenter over HTTPS and TCP/IP on the device's DSL connection to the 2Wire datacenter. The gateway device executes a "bootstrap" command code. The CMS datacenter responds with the device's IP address, DNS information, NAT and DHCP configuration information, and a number of other settings. All of this communication happens between the gateway device and the CMS datacenter invisibly to the end-user.

2Wire's protocol is both efficient and secure for its application. Efficiency is delivered by using a multiplexed sockets manager (MUX). Security is delivered at the network layer by using SSL and at the application layer by using a token-based proprietary authentication technique. 2Wire chose these methods because it owned both the client and server side to each communication. Where the world of Web-enabled applications usually worries about interop-


System Setup - Index - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://gateway.2wire.net/setup/cdless.html Go
Links TM JavaDoc PushToTest Dashboard zope Google email vi Israel News docs NetBeans
Search Pirates! Messenger Bookmarks My Yahoo! Yahoo!
2WIRE
HomePortal 1000S Setup Wizard
HELP EXIT ADVANCED NEXT >
Done Internet

SET UP YOUR HOMEPORTAL 1000S

The Setup Wizard is designed to help you quickly and easily set up your HomePortal 1000S connection to the Internet.

Before you begin, make certain that you have the information provided to you by your service provider. You will need this information for the following installation steps.

To begin the set up of your HomePortal 1000S, enter the 20-character key code that has been provided to you. If you do not have a key code, you can obtain one online at **www.2Wire.com/keycode**.

Key Code: 522P - 22P4 - 5555 - 0000 - F2NV

Click **NEXT >** to continue.

If you are technically advanced, you can manually configure your HomePortal 1000S. Enter in your key code above and click **ADVANCED**.

Figure 13–2 HomePortal configuration begins with a 20-digit keycode. The gateway device provides this Web browser user interface to enter the keycode value. The device then connects to the CMS datacenter to receive its settings, including TCP/IP, DNS, NAT, and DHCP values.

erability and security in an open environment, 2Wire knows the gateway devices would always only communicate with a CMS datacenter. So it only needs to worry about its own interoperability.

Each communication from a gateway device to the CMS datacenter takes three or more steps to authenticate the gateway device, pass a command request and its supporting data, and receive the response and data. This potentially requires many TCP connections for each step. The Internet is already suffering the effects of the HTTP/1.0 protocol, where each step opens a TCP connection for each URI retrieved, costs both transferred packets and round-trip times, and then closes each connection. For small HTTP requests, these connections are a potent source of poor performance because of TCP's slow startup and the round trips required to open and close each TCP connection.

The World Wide Web Consortium (W3C) developed HTTP/1.1 persistent connections and pipelining to reduce network traffic and the amount of TCP overhead caused by opening and closing TCP connections. However, HTTP/1.1 has its own drawbacks for the 2Wire architecture. Consider that HTTP/1.1 pipelining does not adequately support simultaneous rendering of inlined objects or allow for graceful abortion of HTTP transactions without closing the TCP connection. While the W3C was working on its own connection multiplexer (MUX), the work never really took off with developers due to the complexity of its APIs. Details are at http://www.w3.org/Protocols/MUX/.

2Wire engineers chose to use a connection multiplexer (MUX) modeled after Caucho's HMux (details at http://www.caucho.com.) HMux uses a single socket connection for multiple simultaneous conversations that are full duplex streams. Figure 13–3 illustrates a typical MUX-based conversation across a single open-socket connection—of course, the whole point of using MUX is that several of these conversations may be happening simultaneously.

The 2Wire protocol is a variation of the XML-RPC protocol that moves XML data using TCP socket protocols to provide efficiency to each network connection. The gateway device opens a socket connection to the CMS datacenter. The socket connection remains open while several request and response commands process. As opposed to the normal client/host communication, the MUX connection enables the gateway device and the CMS datacenter host to make several simultaneous requests to each other. An XML-encoded payload of data accompanies each command. For example, in Figure 13–2, the first part of the conversation authenticates the gateway device to the CMS datacenter. The XML-encoded payload contains the device keycode. The CMS datacenter authenticates the gateway device and returns an authenticated session identifier to the gateway device.

The 2Wire system uses a command protocol to enable the gateway devices to request updates, receive its initial settings, and provide registration information. The protocol is extensible so 2Wire may add additional commands without redeploying the gateway devices.

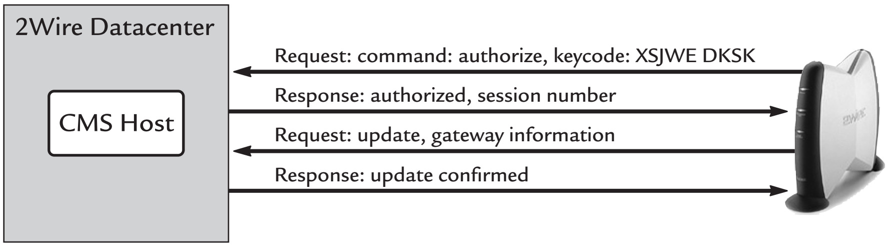
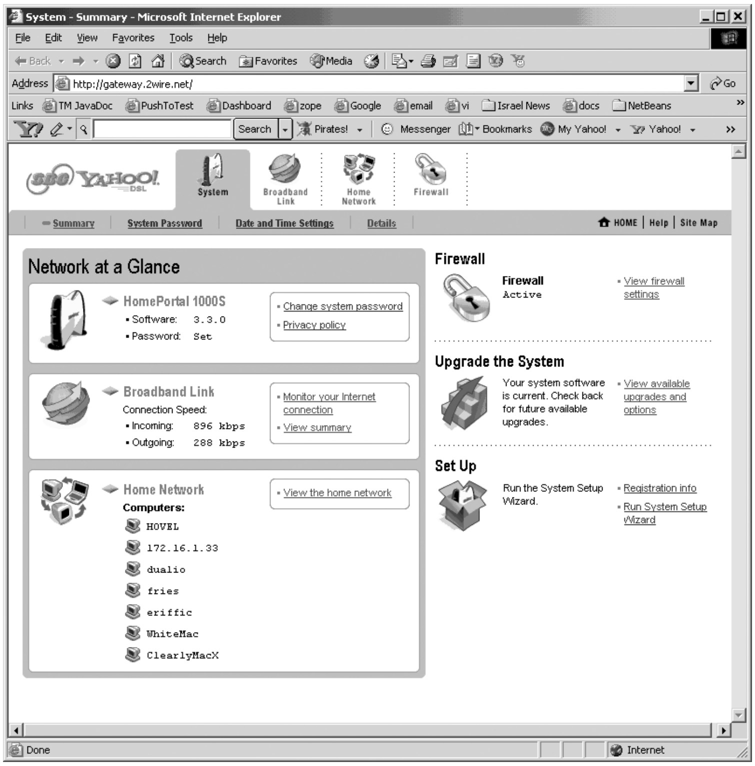

2Wire Datacenter
CMS Host
Request: command: authorize, keycode: XSJWE DKSK
Response: authorized, session number
Request: update, gateway information
Response: update confirmed

Figure 13–3 To more efficiently use network sockets, the 2Wire CMS protocol uses a MUX layer. The gateway box opens a socket and both the gateway and the server make requests over the open socket.


System - Summary - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://gateway.2wire.net/ Go
Links TM JavaDoc PushToTest Dashboard zope Google email vi Israel News docs NetBeans
Search Pirates! Messenger Bookmarks My Yahoo! Yahoo!
SBC YAHOO! DSL
System
Broadband Link
Home Network
Firewall
Summary
System Password
Date and Time Settings
Details
HOME | Help | Site Map
Done
Internet

Network at a Glance

HomePortal 1000S
- Software: 3.3.0
- Password: Set

- Change system password
- Privacy policy

Broadband Link
Connection Speed:
- Incoming: 896 kbps
- Outgoing: 288 kbps

- Monitor your Internet connection
- View summary

Home Network
Computers:
HOVEL
172.16.1.33
dualio
fries
eriffic
WhiteMac
ClearlyMacX

- View the home network

Firewall
Firewall
Active
- View firewall settings

Upgrade the System
Your system software is current. Check back for future available upgrades.
- View available upgrades and options

Set Up
Run the System Setup Wizard.
- Registration info
- Run System Setup Wizard

Figure 13–4 The end-user perspective of the 2Wire system shows a friendly Web browser interface to the HomePortal gateway device settings.

Each of the commands between the gateway device and the CMS datacenter provide the needed interaction to transparently configure and maintain the HomePortal gateway device. This greatly hides the complexity from the end-user. All the end-user sees is the browser-based user interface shown in Figure 13–4. You might wonder, as a consumer of technology devices yourself, why every high-technology device does not have 2Wire's level of ease of use and configuration? In a word: complexity.

Testing in a Complex Environment

Consider what it would take to test the 2Wire system for concurrency problems and to determine the scalability of the system. The system has several potential problem points:

- The communication protocol between the HomePortal gateway device and the CMS datacenter uses a custom authentication mechanism, over a MUX connection, using a modified XML-RPC protocol and a custom command/response protocol. Additionally, the system uses SSL to encrypt the communication between the HomePortal gateway device and the CMS datacenter. There is no off-the-shelf test tool to check this system.
- Since the gateway device should only have to support one user operating its browser-based configuration interface, checking the gateway device for functionality should suffice. However, there is no programmatic way to tell when new functions are added to the device and the dynamically updateable nature of the device means the functions may change without warning.
- The 2Wire datacenter uses redundant load-balanced routers to share support of a population of gateway devices. Load balancers need to be tested to make sure they are sharing the load among the available servers equally.
- The CMS host provides the business workflow needed to support a population of gateway devices. The workflow of any of the gateway device commands requires a transient persistent session to be managed by the CMS host while the MUX connection is open. Poorly implemented code and bugs could potentially introduce concurrency problems into the CMS host that would only be found with loads large enough for the host to run out of resources.
- The CMS host uses a relational database to provide persistent storage of gateway device information. The database needs to be tested for efficiency and capability.

While off-the-shelf testing tools showed facets of CMS performance, no single tool showed the entire picture. And no single tool could answer the overall question of system scalability and performance from end-to-end. 2Wire found in TestMaker the ability to emulate three parts of their system, as illustrated in Figure 13–5.

TestMaker is appropriate for three different tests in the 2Wire system. TestMaker emulates the actions of an end-user behind a browser. TestMaker drives the HomePortal gateway device using native HTTP protocols to test the functions of the gateway device. Second, TestMaker emulates the gate-

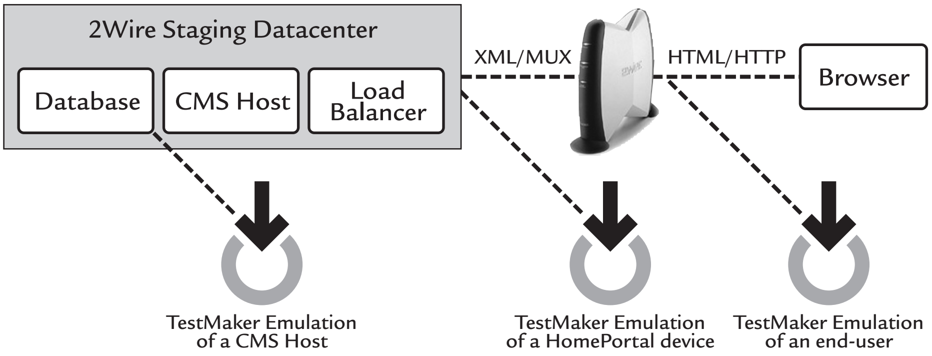

2Wire Staging Datacenter
Database
CMS Host
Load Balancer
XML/MUX
HTML/HTTP
Browser
TestMaker Emulation of a CMS Host
TestMaker Emulation of a HomePortal device
TestMaker Emulation of an end-user

Figure 13–5 TestMaker provides an emulation of the HomePortal gateway device, the CMS host, and even the end user.

way device by interacting with a CMS host using the native CMS host command protocols. Lastly, TestMaker emulates the CMS host by driving the database system using native JDBC drivers.

Additionally, 2Wire saw benefits to how its software development, QA, and IT groups could be leveraged through the use of the TestMaker test environment. 2Wire envisioned these roles:

- **Software Development Group**—writes Java modules that plug into the TestMaker framework that implement 2Wire Gateway functions. Development writes test agents that call Java modules to perform functional tests against new 2Wire CMS functions.
- **Quality Assurance Group**—writes test agent scripts to run performance and scalability tests of new 2Wire CMS functions. These tests isolate performance bottlenecks and show where optimizations will lead to better throughput, better resource usage, and stability.
- **Operations Group**—IT managers use the test agents from development and QA as proof-of-delivery checks on new 2Wire CMS builds and also as a quality-of-service monitor.

While 2Wire was concerned with the potential problem points listed above, their highest priority was to certify the CMS host as ready to serve a population of gateway devices. So, 2Wire contracted with PushToTest to provide a custom test environment based on the TestMaker open source project and to conduct an independent audit of CMS system performance and scalability:

- We wrote a scalability study proposal to define the criteria for good performance and scalability of the backend HomePortal/ CMS support system. We interviewed the software developers, QA technicians, and IT managers to understand each group's goals and concerns. The resulting test criteria define four user archetypes—prototypical HomePortal users—from which we modeled intelligent test agents. The criteria define a combination of agents to simulate user demands on the system.
- We built, deployed, and then analyzed an intelligent test agent system to determine the maximum concurrent users and throughput measured in transactions per second and how these relate to the number of HomePortal gateway devices supported by a server.
- We wrote the XML-RPC protocol handlers needed to simulate a HomePortal unit. PushToTest then used TestNetwork, the commercial version of TestMaker that features a distributed test and analysis environment, to run thousands of agents concurrently to simulate a real-world environment to determine system functionality, scalability, and performance. Details on TestNetwork are at http://www.pushtotest.com/ptt/TestNetwork/index.html.
- We trained 2Wire software developers and QA technicians on using the PushToTest TestMaker and TestNetwork software so that 2Wire engineers will be able to build new tests and maintain existing tests.

The project resulted in test data that 2Wire takes to its customers, including SBC and Verizon. Plus, the extended TestMaker and TestNetwork environment provides an extensible test platform for checking system scalability, performance, and functionality as the infrastructure, CMS software, and deployments change.

Next this chapter describes the test method, how the criteria for good performance were developed, the test environment, and the test results.

The Test Method

2Wire designs CMS solutions to be reliable and scalable in increasingly large deployments. While the individual 2Wire system components are standard off-the-shelf proven Web infrastructure, 2Wire needed to develop and use new test methodologies, techniques, and tools to understand the overall system performance. To address the advanced nature of the 2Wire CMS solution design, 2Wire developed two test methodologies:

- **User goal-oriented testing**—Rather than testing individual system functions, 2Wire developed user archetypes for prototypical users. Intelligent test agents were coded against the user archetypes. Test agents may run concurrently and each logs its results for later analysis. The tests measure system performance against user goals.
- **Real-world device emulation**—The test environment implements the native protocols used by the CMS datacenter and HomePortal gateway devices. Consequently, the tests drive the entire system as a real gateway device would. Additionally, the emulated devices may be run from within the datacenter or from remote locations.

2Wire's test method measures system scalability and performance in two ways:

- **Throughput**—Measures the capacity of a CMS server to efficiently handle each gateway device requests. A system scales well vertically when all requests are handled within a predetermined duration.
- **Scalability**—Measures the capacity of the CMS host datacenter to respond to increasingly large populations of gateway devices. A system scales well horizontally when overall system performance is not impacted as additional client devices are added.

By design, CMS solutions provide a polling-style system where CMS responds to requests made by gateway devices. The system uses Internet protocols (XML-RPC over TCP/IP connections) for the gateway devices to interoperate with the CMS host. These protocols are used in three increasingly complex categories of transactions. These are outlined in Table 13–1.

Table 13–1 Categorizing Transactions by Payload Size

Category	Description	Average payload size
Category 1	Services that require stateless simple responses to requests. For example, updating DNS services.	1,200 bytes
Category 2	Services that require multiprotocol multistep stateful transactions with database access. For example, the HomePortal gateway devices issue update requests to the CMS host periodically.	4,500 bytes
Category 3	Services that require multistep transactions that require large amounts of data transfer. Examples: CMS provides automatic upgrade capabilities to the gateway devices.	75,000 bytes

By design, the 2Wire gateway devices have a normal behavior that includes periodic communication with the CMS host, regular user requests for help, and occasional software upgrades. For the purpose of the test project, we made the assumptions about the number of requests a 2Wire HomePortal gateway device generates requests for the operations listed in Table 13–2.

Table 13–2 CMS Operations and Planned Frequency

Operation	Planned frequency
Operation #1, the HomePortal device checks with the CMS host for updates.	1 per day
Operation #2, the HomePortal device receives its initial operational settings and updated programming.	1 on startup, then 1 per year
Operation #3, the HomePortal user requests information about settings.	1 per month
Operation #4, the HomePortal receives new programming.	3 per year
Operation #5, the HomePortal registers itself with the CMS host.	1 per year

2Wire scalability and performance tests emulate the real-world CMS host datacenter serving a population of HomePortal gateway devices by implementing each operation (Heartbeat, Bootstrap, etc.) in an intelligent test agent. A mixture of test agents issues live requests to a CMS test host datacenter concurrently. For example, when testing a CMS solution with an emulated population of 5,000 gateway devices, the concurrently running test agents issue the following mixture of operations listed in Table 13–3.

Table 13–3 Mixture of Planned Operations

Operations	Percentage mix	Operations issued from the test agents
Operation #1	96.960%	4,848
Operation #2	0.177%	9
Operation #3	2.155%	108
Operation #4	0.531%	27
Operation #5	0.177%	9

The intelligent test agent method provides an easy technique to change the percentage mixture of operations. Additionally, testing with larger populations of gateway devices is accomplished by adding more concurrently running test agents.

While the TestMaker test agents create near real-world usage, there are setup operations that need to be run prior to starting a test. Figure 13–6 shows the order in which the steps are taken. During the setup phase a special configuration test script creates a delimited data file of unregistered gateway devices by connecting to the CMS database. During the Live Test phase the agents speak native CMS protocols to drive transactions. The analysis phase tallies the results recorded during the Live Test phase into statistical reports.

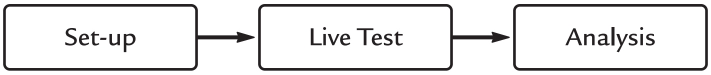


Figure 13–6 The steps to conduct a test.

Test results are stored in delimited log files and analyzed for transactional statistics, including TPS for successful operations and a weighted SPI that includes operations that resulted in errors. SPI, first introduced in Chapter 2, is a weighted measurement that accounts for slow performance in individual requests and errors during each test run.

Let's look at how a test is staged using this methodology.

Test Environment Considerations

With the extended TestMaker enabled to speak native CMS protocols, 2Wire has a lot of flexibility to staging scalability and performance tests. For example, test agents may be run from within the datacenter to conduct functionality tests, and external to the datacenter for scalability testing. Figure 13–7 shows a configuration where test agents are run outside of the datacenter to test overall system scalability under increasing numbers of emulated HomePortal gateway devices.

The TestNetwork environment enables 2Wire to develop system load in near real-world conditions to make the results as meaningful as possible. The TestNetwork environment uses TestMaker as a central console to coordinate the launch of a test on a set of TestNodes. Because of the overhead of the

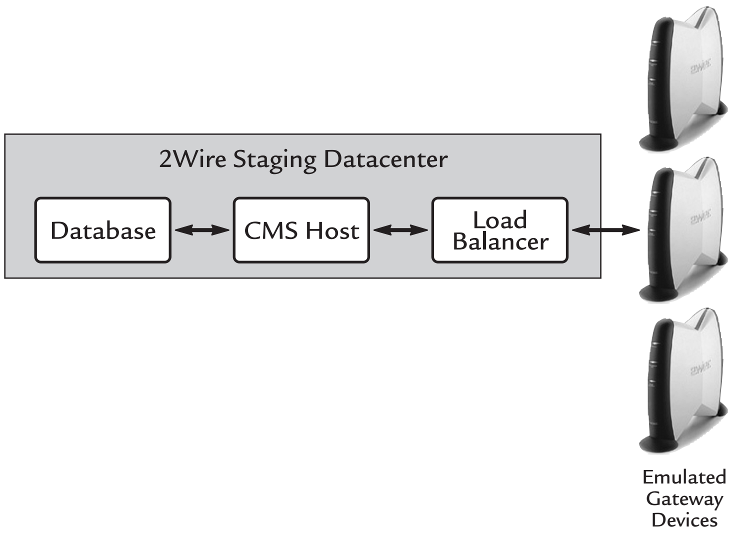


Figure 13–7 A scalability testing configuration checks the CMS datacenter for throughput by increasing the numbers of emulated HomePortal gateway devices.

gateway device emulation, each TestNode emulates up to 150 gateway devices. The TestNodes return their logged results data to TestMaker for analysis. Table 13–4 explores the devices, tools, and settings for the test environment in more depth.

Table 13–4 Test Environment Devices, Tools, and Settings

Component	Description
Intelligent test agents	Agents implement the behavior of prototypical users. For example, by speaking native CMS protocols an agent may emulate a HomePortal gateway device. Running multiple concurrent agents simulates multiple gateway devices.
TestNetwork TestNodes	Each TestNode is an inexpensive RedHat Linux-based system with a 1.4 MHz Intel processor, 256 Mbytes of memory, and IDE hard disk mechanisms. TestNodes run the Java Virtual Machine (JVM) version from Sun and the TestNetwork software, including the HomePortal gateway device native protocol emulation.
2Wire Production Datacenter, Staging Server	2Wire operates a production datacenter that features staging servers that are used to test new CMS software prior to installing the new software onto the live production servers. Staging servers are equal to the production servers in terms of bandwidth and capacity.
Network connectivity	Test agents interoperate with the datacenter over switched 100 Mbit Ethernet connections. TestNode devices operate on the company network and are isolated to a small subnet.
Firewall	TCP/IP connections between TestNodes and CMS hosts pass through a firewall device.

With this configuration for the test environment, 2Wire conducts scalability and performance tests in a laboratory setting and extrapolates to near real-world results. The team took the following steps to conduct a test.

Between individual tests the TestNetwork/Gateway emulation and the application server/database will be restarted to reduce the risk that memory leaks and unreleased network connections might affect the test results.

1. Generate gateway identification numbers for each TestNode. A special-purpose test script queries the CMS database for available serial numbers. The script divides the ID numbers into equal groups according to the number of TestNodes to be used. The script copies the resulting ID number files to the TestNodes.
2. Determine maximum throughput of an individual TestNode. A special test script runs increasing levels of concurrent test agent threads on a local TestNode. The script determines the most efficient amount of concurrent test agent threads by evaluating throughput against the number of threads. At the time, this script determined that each TestNode could handle 75 concurrently running test agent threads.
3. Determine maximum throughput of a single CMS host. Perform several runs of concurrent test agents, starting with 50 agents running on each TestNode and between runs increasing by 10 agents on each TestNode. Each run will take 3 minutes, after the run-up time it takes to instantiate the agents.
4. Analyze the results to find the maximum TPS level with the least amount of exceptions. The agent script automatically outputs a comma-delimited log of results. The *tally_agent* script then analyzes TPS values from the logs.
5. Run steps 3 and 4 again, but this time run the test for 15 minutes. This was done to find memory leaks, unreleased resources, and database resource usage (CPU, memory, disk). Later we ran the same test for 8 hours. We then analyzed the differences between the short-duration and longer-duration runs.

Using the test environment and these steps, 2Wire can change the nature of what is tested by adding test agents and adjusting the duration and makeup of each test.

Constraints in the Test Environment

So far in this chapter, it may appear that the TestMaker test environment is all things to 2Wire's effort to check its own technology for scalability, performance, and functionality. TestMaker is a good start, but it is not thorough. This section describes some of the things to look out for when testing in your own environment.

All of the test results 2Wire provides to its customers include a general disclaimer that *your tests may vary.* Changes in equipment, network connectivity, and software will greatly impact test results. Because of this, 2Wire offers PushToTest services to its customers to verify a customer's test goals, methodology, test configuration, and results. Additionally, 2Wire engineering constantly strives to improve its software performance and reliability through regular maintenance and upgrades. Each new 2Wire software rollout is intended to positively affect scalability and performance results.

Tests conducted in a laboratory setting may not be valid to extrapolate into real-world production environments where network latency, TCP/IP connectivity, and routing issues may put demands onto the system that are not in the test environment.

2Wire's goal in testing CMS included measuring the maximum concurrent HomePortal gateway device to CMS datacenter transactions. The tests did not include all of the gateway device commands. For example, the gateway device provides dynamic DNS service and FTP service that was not tested. Additionally, the tests performed minor checks of the response data for validity and integrity. If the response contained a *succeeded* response code, then that is all the test agents looked for in a successful response. A more in-depth test would also check the response data for validity.

In addition to the end-to-end tests, 2Wire's IT managers wanted to do component isolation tests of segments of the network infrastructure. For example, they wanted to bypass the load balancer and run a battery of tests directly against a particular CMS host and database server. They sought to get a more detailed statistical view of the components of their infrastructure. Unfortunately, we just did not have enough time to do component isolation tests.

I encountered another problem when moving the TestMaker environment from Windows-based platforms to Linux, and also when moving from one Java Virtual Machine to the next major version. It turns out that the default encoding styles for URL and string encoding changed between platforms.

Summary

Using the TestMaker tool and the methodology presented in this book, 2Wire better understood its ability to serve customers. The knowledge gained through the experience enabled 2Wire to save money by buying only the network and server infrastructure it needed, and to certify their datacenter as ready to serve large populations of 2Wire HomePortal gateway devices. The resulting custom test and monitoring environment checks 2Wire systems for scalability, reliability, and availability. The new test environment paid for itself in its first month of operation by leveraging the development, QA, and operations efforts together.

14 Making the Right Choices for SOAP Scalability

Software developers live in a time that offers the greatest choice of software development tools, application servers, and connectivity ever. Each choice that you make affects the scalability and reliability of your finished application, especially if you're building Web services. For example, as you will learn later in this chapter, my study of SOAP encoding styles found a 30-fold performance improvement by choosing one SOAP encoding style over the others. By understanding the performance impact of SOAP encoding styles, Web service development tools, application servers, and platforms, our choices greatly improve system performance.

This chapter presents results of an investigation that shows how each choice immediately impacts scalability and reliability. It discusses the impact of letting development tools make choices for us on scalability and performance. Then it presents directions, tools, and test agents to stage tests in your own environment.

Why is SOAP So Popular?

In my experience, when independent technology innovations intersect, the world enjoys life-changing products, services, and techniques. For example, the light bulb required both electricity generation and filament fiber technology. In the case of Web services, enterprise information technology managers had just come off a multiyear binge where they bought huge numbers of computers, servers, routers, and other Web infrastructure. Left with a recession, terrible stock market, uncertainty about the world economy, and having to contend with terrorism and SARS, these information managers returned to their existing Web infrastructures to increase productivity of their teams through new software projects.

At the same time, most software developers realized they really liked XML. For example, XML was much better than using the Microsoft Windows Registry or text-based property files to store and describe application data. Software developers saw a good thing in XML and wanted to find more ways to use XML in their applications.

As a software developer I began noticing APIs that expected to receive a value that contained XML encoded data. For example, when building a portal system for Sun Microsystems, I found that the servlet to create a new user account received the fields that made up the user contact information (email, address, and telephone number) in an XML document. Rather than pass in one value at a time to a method, instead the method took one XML value that contained several values. Using XML to implement an application's interfaces is a clear win to developers. Plus, these XML-described interfaces could work across platforms and programming languages. With XML everything looks like an interface.

These intersecting technologies power the widespread enthusiasm for Web services. At the same time, software developers were again experimenting with software architectures, especially with the location of application business logic and presentation code. Presentation code handles windows, mice, keyboard, and other user interactions. Business logic is the instructions that define the behavior and operation of an application.

The first-generation software architecture built the presentation and business logic on a single system. In the second generation, client/server architecture brought back the large, centrally controlled datacenter so familiar in the 1960s when mainframes ruled the information world. In client/server architecture, the desktop system is a "dumb" terminal that only needs to display the data provided by the server. The early Internet was modeled after client/server architecture where the browser made simple requests to a server. As browser's improved in functionality—applets, JavaScript, ActiveX, and DHTML were introduced—some systems included business logic on the desktop side. However, the majority of function remained on the server.

The age of "grid computing" is upon us where an application hosts business logic modules on the desktop or server. The modules discover each other using UDDI and P2P technologies. Also multiple copies of the business logic modules may run in a grid of datacenters to allow failover, dynamic routing, and functional specialization. All of these architectures run in a Web environment and can host Web services.

So even if SOAP-based Web services are replaced with some other type of Web service technology, remote procedure calls using XML encoded data will be around for a very long time.

SOAP Encoding Styles

SOAP uses XML to marshal data that is transported to a software application. Most of the time, SOAP moves data between software objects, but the SOAP specification was intended to be useful for old legacy systems as well as modern object-oriented systems. Consequently, SOAP defines more than one data encoding method to convert data from a software program into XML format and back again. The SOAP encoded data is packaged into the body of a message and sent to a host. The host then decodes the XML-formatted data back into a software object.

Since SOAP's introduction, three SOAP encoding styles have become popular and are reliably implemented across software vendors and technology providers:

- *SOAP RPC encoding*, also known as Section 5 encoding, as defined by the SOAP 1.1 specification and later defined in SOAP 1.2 as RPC encodings and conventions.
- *SOAP Remote Procedure Call Literal encoding (SOAP RPC-literal)* uses RPC methods to make the call but uses an XML do-it-yourself method for marshaling the data.
- *SOAP document-style encoding*, also known as message-style or document-literal encoding.

There are other encoding styles, but software developers have not widely adopted them, mostly because their promoters disagree on a standard. For example, early on in the invention of Web services, Microsoft promoted Direct Internet Message Exchange (DIME) to encode binary file data, while the rest of the computer industry adopted SOAP with Attachments. SOAP RPC encoding, RPC-literal, and document-style SOAP encoding have emerged as the encoding styles that a software developer can count on.

Some developers do not realize that such encoding styles exist because the tools they use to develop Web services are doing the work of implementing the encoding styles for the developer. For example, BEA WebLogic Workshop provides a fast and efficient implementation of the JWS interface. JWS implements a set of APIs and a standard description of the files the JWS engine needs to automatically deploy the Web service on the server. JWS builds the Web service deployment descriptors for you automatically. So you need to only define the public Java methods and JWS publishes the SOAP proxy to access the methods. This makes development appear very easy, but there is a lot of work going on under the covers, so to speak. We'll see this in more depth later in this chapter.

Before I discuss SOAP encoding style's impact on performance, you should understand the differences between these styles of SOAP encoding. Figure 14–1 shows the entire stack for a SOAP RPC encoded call.

SOAP RPC is the encoding style that offers the most simplicity for developers. The developer makes a call to a remote object, passing along any necessary parameters. The SOAP stack serializes the parameters into XML, moves the data to the destination using transports such as HTTP and SMTP, receives the response, deserializes the response back into objects, and returns the results to the calling method. Whew! SOAP RPC handles all the encoding and decoding, even for very complex data types, and binds to the remote object automatically.

Now, imagine you are a developer with some data already in XML format. SOAP RPC also allows literal encoding of the XML data as a single field that is serialized and sent to the Web service host. Since there is only a single parameter—the XML tree—the SOAP stack only needs to serialize one value. The SOAP stack still deals with the transport issues to get the request to the remote object. The stack binds the request to the remote object and handles the response.

Lastly, in a SOAP document-style call, the SOAP stack sends an entire XML document to a server without even requiring a return value. The message can contain any sort of XML data that is appropriate to the remote service. In SOAP document-style encoding, the developer handles everything,

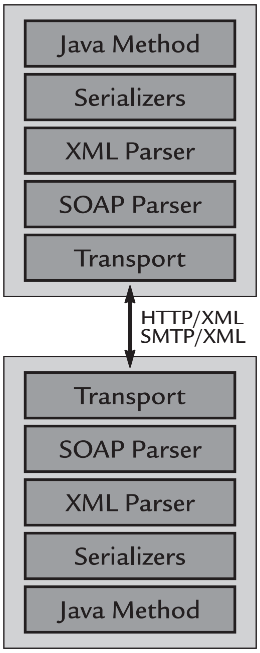


Figure 14–1 A Java method calls a Web service by using the SOAP stack and SOAP RPC encoding.

including determining the transport (e.g., HTTP, MQ, SMTP), marshaling and unmarshaling the body of the SOAP envelope, and parsing the XML in the request and response to find the needed data.

The three encoding systems are compared in Figure 14–2.

SOAP RPC encoding is easiest for the software developer; however, all that ease comes with a scalability and performance penalty. SOAP RPC-literal encoding is more involved for the software developer to handle XML parsing, but requires fewer overheads from the SOAP stack. SOAP document-literal encoding is most difficult for the software developer, but consequently requires little SOAP overhead.

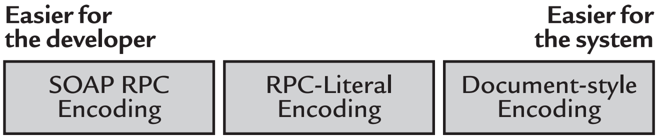


Figure 14–2 SOAP encoding styles offer software developers greater productivity, but it comes at a performance and system resource cost.

Why is SOAP RPC easier for the developer? With this encoding style, you only need to define the public object method in your code once; the SOAP stack unmarshals the request parameters into objects and passes them directly into the method call of your object. Otherwise, you are stuck with the task of parsing through the XML tree to find the data elements you need, and then you get to make the call to the public method.

There is an argument for parsing the XML data yourself: Since you know the data in the XML tree best, your code will parse that data more efficiently than generalized SOAP stack code. As we will see when we measure scalability and performance in SOAP encoding styles, we will find this to be the case.

But before I go further into that, we look at how enterprise information systems managers are coming to grips with SOAP encoding styles and scalability.

Simple Object Access Needs Simple Testing

Elsevier (www.elsevier.com) is the leading research content publisher for the science, technology, and medical industries. Elsevier now uses a content publishing platform that uses SOAP to build application programming interfaces. Elsevier's information managers need to know if their choices of SOAP encoding style will scale and perform to handle millions of transactions every day. Their decisions affect how Elsevier will invest capital in new infrastructure. Over time, they need to know how new releases of their own software, new releases of application server software, and platform changes will affect scalability and performance.

Elsevier learned about TestMaker through the open source community and contacted PushToTest (details at www.pushtotest.com) to see if TestMaker was appropriate for their testing needs. Elsevier asked PushToTest to conduct an independent audit of SOAP stacks and encoding styles to answer their questions about system performance and scalability. The resulting test environment is illustrated in Figure 14–3. PushToTest delivered a Test Web Service (TWS) that handles RPC, RPC-literal, and document-style SOAP messages and runs on a variety of application servers. The environment is completed with a set of intelligent test agents to check TWS for scalability and performance.

TestMaker checks Web services for scalability, performance, and reliability. Software developers, QA analysts, and IT managers use TestMaker to

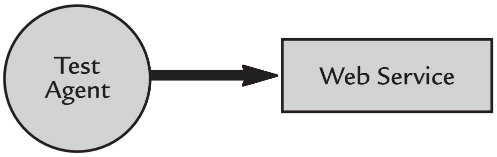


Figure 14–3 The test environment to check SOAP scalability uses test agents to make requests to a SOAP-based Web service.

build intelligent test agents that implement archetypal user behavior. The agents drive a Web service using native protocols (HTTP, HTTPS, SOAP, XML-RPC, SMTP, POP3, IMAP) just as a real user would. Running multiple intelligent test agents concurrently creates near production-level loads to check the system for scalability and performance.

In addition to checking SOAP encoding scalability, the Elsevier test environment provides a benchmark specific to Elsevier's systems to show a performance comparison for a variety of application servers and platforms. For example, TWS is currently implemented to run on IBM WebSphere, BEA WebLogic Workshop, and the SunONE application server. I am confident that ports to webMethods Glue, Apache Axis, Systinet WASP, and other application servers is straightforward.

I built the Elsevier test environment by customizing TestMaker to support SOAP RPC, SOAP RPC-literal, and SOAP document-style requests, and by implementing TWS to respond to requests in these encoding styles. The request to TWS contains two parameters: the first defines the size of the response and the second defines a `delay` value before responding. TWS responds by creating a response document containing random gibberish words that appeared in five response elements; each element has one child element. A TestMaker test agent uses the Apache SOAP library to make requests to TWS. The test agent varied the number of concurrent requests to TWS and the payload size of the response. The test agent logged the results to a delimited log file, which was subsequently summarized by a tally script. The tally script determined the number of TPS performed by the test by counting the duration of successful transactions. We defined success as the absence of transport or SOAP faults.

With Sun Microsystems support, I ran the tests on Sun Solaris E4500 servers with six CPUs and 4 GB of RAM. The TWS used the SOAP stack provided by the underlying application server. For example, WebSphere provides Apache SOAP, BEA WebLogic provides their own implementation that uses the JAX-RPC APIs, and the SunONE application server uses the Java 1.4 JAX-RPC library. On the client side TestMaker uses the Apache SOAP library.

In the Elsevier project, I found that a developer's choice of encoding style determines to a large extent the scalability and performance of a Web service. The SOAP implementations universally showed scalability problems when using SOAP RPC encoding, especially as payload sizes increased, as illustrated in Figure 14–4.

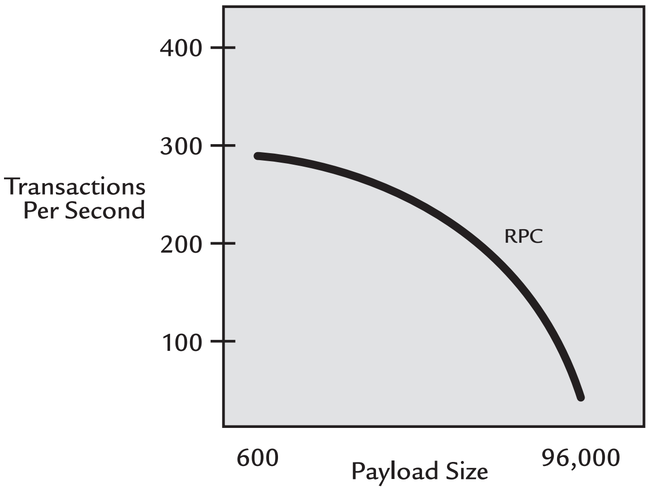


Figure 14–4 SOAP RPC encoding: Scalability problems become noticeable with increased payload sizes.

As Figure 14–4 shows, the test agent recorded 294 transactions per second when making requests where the response SOAP envelope measured 600 bytes of SOAP RPC-encoded data. As the test agent increased the response size, the transactions per second plummeted. When making requests of 96,000 bytes of SOAP RPC-encoded data, the agent measured only 9.5 transactions per second.

When the test environment uses SOAP document-style encoding, the performance fared much better, as you can see in Figure 14–5.

With 600 bytes of document-encoded data, the test agent measured 469 TPS. Recall that the SOAP RPC-encoded requests gave us 294 TPS for requests of the same size. Additionally, when the test agent increased the

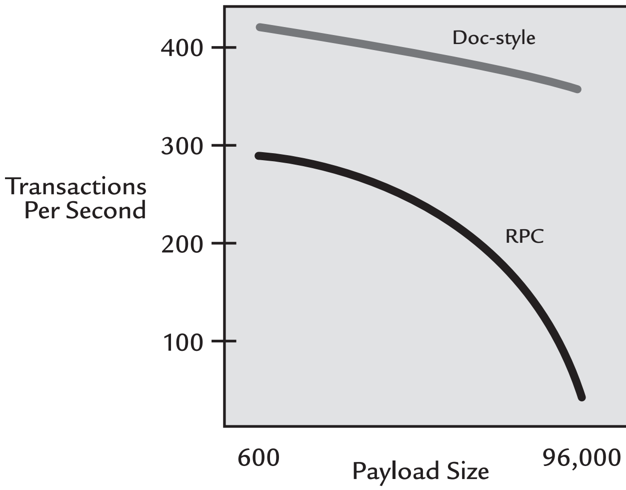


Figure 14–5 Document-style encoding: Performance stays relatively stable with increased payload sizes.

response size, the TPS values did not degrade significantly when we used document-style encoded responses.

When the test environment uses SOAP RPC-literal encoding, I found an efficient middle ground, as you can see in Figure 14–6.

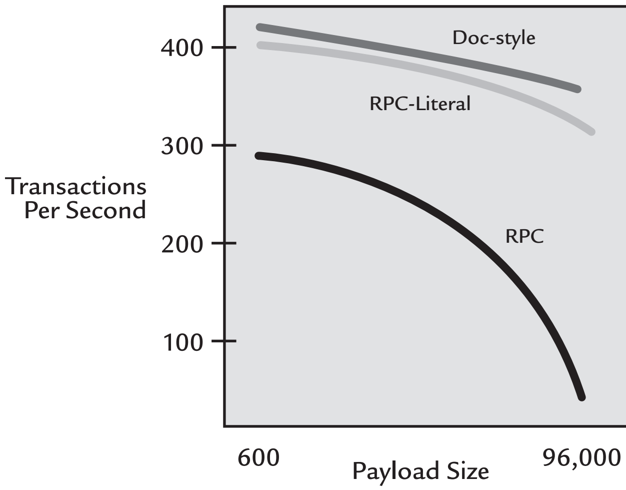


Figure 14–6 SOAP RPC-literal provides the performance benefits of SOAP document-style encoding with a little more work required to parse through the XML data.

Should You Let the Tools Do the Driving?

While the SOAP encoding styles provide a good range of power and flexibility, they also introduce interoperability problems. Most of the SOAP tools on the Java platform default to SOAP RPC encoding styles. For example, when using IBM WebSphere Application Developer, the default encoding style is set to SOAP RPC. On the other side of the divide, .NET development tools implement document-style SOAP calls by default. This is akin to watching two boats pass in the night. Both can be made to interoperate, but developers need to be wise to the different encoding styles to avoid problems.

In their attempt to make software developers' lives easier, these tools may be making decisions for you that affect scalability. This was highlighted when Microsoft and Sun debated the relative virtues of J2EE and .NET at an event hosted in Silicon Valley by the Software Development Forum. Details are found at http://www.sdforum.org. Microsoft made the argument that it serves developers best by being the sole supplier of a complete solution. On the other end of the spectrum, Sun posited that developers should have a choice of tools that they can assemble into a solution. This top-down versus bottom-up argument permeates into both companies' development tools. For example, representatives from Sun and Microsoft were asked to explain why developers would choose SOAP RPC encoding over SOAP document-style encoding. Microsoft's reps gave a somewhat technical answer, but conceded that they thought the issue was moot, since developers should rely on their development tools to make decisions of encoding styles.

Software developers serve themselves best by making informed decisions of how helpful their development tools and environments should be. Understanding each tool's handling of SOAP encoding styles is an important factor to delivering well performing and reliable software projects.

May I Freak Out Now?

With 600 bytes of SOAP RPC-literal encoded data, the test agent measured 422 TPS. That is nearly the performance recorded for SOAP document-style requests. SOAP RPC-literal encoding did not show the plummeting TPS function of SOAP RPC encoding performance.

At this point your mind might be racing with questions like:

- Is this true? Can SOAP scalability be that bad?

- What are the performance differences between the application servers? Will IBM WebSphere perform better than BEA WebLogic?
- What are the most important considerations to build reliable Web services?
- What equipment do I need?
- What is the best way to deploy Web services?
- What is the best software platform to build my Web service?

In my experience every production environment is unique. So, rather than try to be your answer guy for every application server and encoding style, I would like to give you a performance kit that you can download and use in your own production environment. I have made a generalized version of the Elsevier test environment available for free download for your immediate use at http://www.pushtotest.com/ptt/kits/encodingkit.html. In the next section I present the kit and show how you may use it in your own environment.

The Performance Kit

Enterprise information manager choices for Web service infrastructure are critical to deploying well performing, highly reliable, and scalable systems for running Web services. I receive calls everyday from software developers, QA technicians, and IT managers wanting to know:

- What are the most important considerations to build reliable Web services?
- What equipment do I need?
- What is the best way to deploy Web services?
- What is the best software platform to build my Web service?

Several things prevent me from having an immediate answer to your questions. These include the following:

- The application servers you use are a moving target. The popular commercial and open source software vendors work tirelessly to improve performance, add functions, and improve techniques to use their products and technology. The performance and scalability characteristics change with each new version of their products.

- The underlying code libraries in a SOAP stack are moving targets too. For example, most of the Java-based Web service applications use the popular Apache Xerces library to parse XML data. Xerces is both complex and slow. Changing to a newer version of Xerces or replacing it with another library greatly impacts Web service performance and scalability.
- Your network infrastructure (load balancers, routers, server equipment, storage devices) is unique from everyone else. This makes an apples-to-apples comparison very difficult.
- Your own software development efforts will improve performance and scalability as new builds of your software application are applied to your production environment.

So rather than try to answer your questions, this section shows how to understand Web service scalability and performance in your own environment.

The Web Services Performance Kit helps information managers with reliable and repeatable performance measurement tools for systems that drive Web service-based business. The kit may be applied to any Web service-based system. However, the examples are specific to the SunONE application server, BEA WebLogic server, and IBM WebSphere application server. First I give an overview of the kit's contents and then we will see how the kit is applied to an application server environment.

How Do I Get the Performance Kit?

The Web Services Performance Kit is available for free download at the PushToTest Web site. Point your favorite browser to http://www.pushtotest.com/ptt/kits/index.html to download the kit. The PushToTest site features the latest software, archives of previous versions, frequently asked questions about the kit, and various technical support services, including email support lists for users and developers.

The kit is distributed in a Zip archive file format. Extracting the file contents installs everything in the kit. You will need to also download TestMaker from the PushToTest site. TestMaker is fully described in Chapter 5. The kit contains test agents that run in the TestMaker environment and the Test Web Service that runs everywhere Java 1.4 or greater runs, including Windows, Linux, Solaris, and Macintosh OS X. Other Java-enabled platforms are capable of running the kit but have not been tested.

TestMaker updates appear generally once a month to offer bug fixes and new features. The PushToTest Web site offers a free email announcement service to send email alert messages when new versions become available. (PushToTest does not publish or share email addresses.)

Installing the Performance Kit

Installing the kit is very simple. Balancing, that is, the Test Web Service, which is often very difficult to install and configure. Inside the Performance Kit you will find the following things.

- Easy-to-read instructions describing how to build and test Web services on the SunONE application server, BEA WebLogic server, and IBM WebSphere server. In this chapter, due to space considerations, I limit the examples to testing on a BEA WebLogic Workshop.
- TWS—a free, open source Web service that simulates a database-driven Web service. TWS responds to a variety of SOAP encoding styles and on several application servers.
- Performance_Agent—a free intelligent test agent that will determine the scalability and performance of the TWS. Performance_Agent makes multiple concurrent requests to TWS and measures performance in a transactions-per-second result.
- TestMaker—a free, open source utility to run Performance_Agent and to create agents for your own Web service testing. See Chapter 5 for details on TestMaker.

The Performance Kit comes as a compressed archive in ZIP format of HTML documents, test agent script files, and Java source code files. Please use a decompression utility to extract the files onto your local computer. The index.html document, among the decompressed files, contains installation, configuration, and "how-to" instructions for the kit.

PushToTest provides a variety of support options to help you use the kit. Support is offered for free from the PushToTest community. Additionally, for-pay support agreements are offered. Details on both are at http://www.pushtotest.com/ptt/support.html.

Getting Started

After you unpack the kit, I recommend you spend a few minutes getting to know its components. Take a look at the documentation and peruse the source code files. Most of the code is self-documented. Then follow these steps to run the kit:

1. If you have not already done so, download and install one of the supported application servers. Sun, BEA, and IBM feature free trial versions of their software: SunONE application server, BEA WebLogic server, and IBM WebSphere application server. This chapter looks specifically at BEA WebLogic Workshop.
2. Install the Test Web Service.
3. Download and install TestMaker.
4. Run the PerfCheck agent to test TWS for performance. PerfCheck is an intelligent test agent written and run with TestMaker. PerfCheck includes a Tally script that shows the TPS and other performance metrics for the TWS.

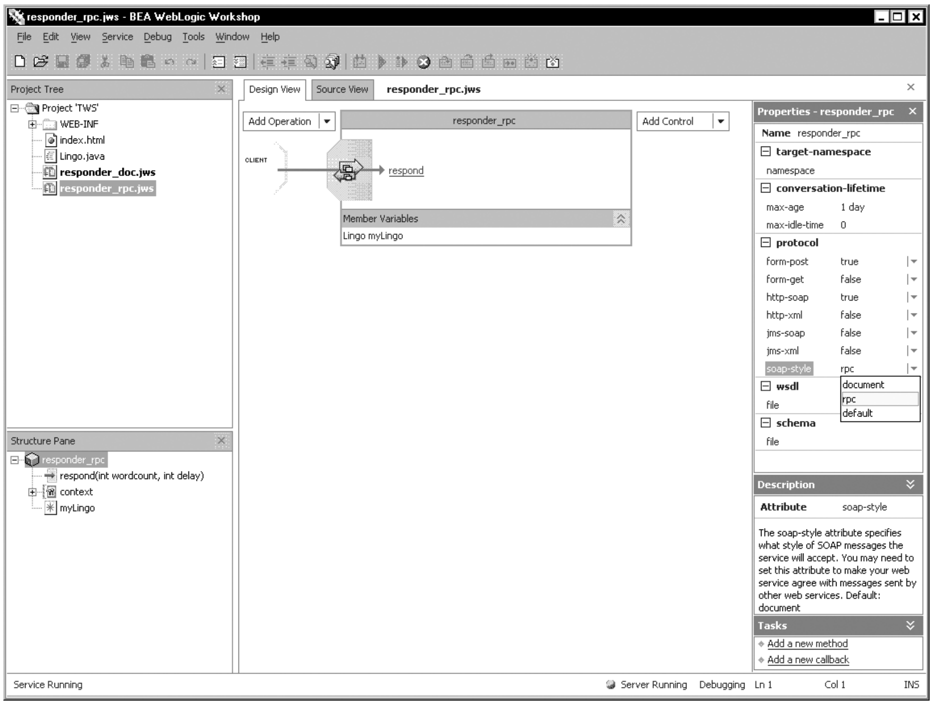
Next we look at the components of the kit and see what steps you should take to install, configure, and run the kit.

The Test Web Service

The TWS is a Web Service that simulates a typical database-driven Web service. TWS takes SOAP RPC encoded requests and returns a response. The response contains gibberish words of a length determined by a parameter of the request. TWS is a simple and easy choice to test application server platforms for performance, reliability, and scalability.

The kit implements TWS on a variety of application servers. For BEA WebLogic, I used BEA's integrated development environment named Workshop. Workshop's designers have figured out a clean way to present the myriad of options found in the SOAP and WSDL specifications. To see this in action, look at Figure 14–7.

Public methods in a Java class are turned into SOAP-based Web services automatically. For example, Figure 14–7 shows the `responder_rpc` class's single public method named `respond`. The right-most Properties pane shows the SOAP options for respond. Figure 14–7 shows how easy it is to change



Properties - responder_rpc

Name responder_rpc

target-namespace
namespace

conversation-lifetime

max-age	1 day
max-idle-time	0

protocol

form-post	true
form-get	false
http-soap	true
http-xml	false
jms-soap	false
jms-xml	false
soap-style	rpc

document
rpc
default

wsdl
file

schema
file

Description

Attribute soap-style

The soap-style attribute specifies what style of SOAP messages the service will accept. You may need to set this attribute to make your web service agree with messages sent by other web services. Default: document

Figure 14–7 Workshop makes the many SOAP options visually easy to adjust, including setting SOAP encoding styles between document-style and SOAP RPC-style.

SOAP encoding styles for the `respond` method. The choices are document, rpc, and default. WebLogic's default value chooses SOAP document-literal style encoding.

Choosing a SOAP encoding style in the Design View changes the JWS header in the Java code that is viewable in the Source View. JWS does the work for you. For example, the `respond` method in the Source View is declared by default as:

```
import weblogic.jws.control.JwsContext;

/**
* @jws:protocol form-get="false" soap-style="document"
*/

public String respond(int wordcount, int delay) { ...
```

This is enough for Workshop to package the deployment descriptors to tell the JWS engine to deploy a Web service that has a `respond` method that uses SOAP document-style encoding.

When we change the encoding style in the Design view to rpc, the Java code changes to:

```
import weblogic.jws.control.JwsContext;

/**
* @jws:protocol form-get="false" soap-style="rpc"
*/

public String respond(int wordcount, int delay) { ...
```

This does not appear to make much difference in the code; however, behind the scenes a lot is happening. And much of it impacts the scalability of the `respond` method. When this Java code is run on a JWS-compatible virtual machine—such as WebLogic Workshop—then the virtual machine automatically adds a SOAP proxy to the method so it will respond and reply to SOAP requests. The proxy handles all of the marshaling and unmarshaling of parameters to make the method call and respond with a response value.

To install and run TWS in WebLogic Workshop, run Workshop, open the TWS project, open the *responder_rpc.jws* file, and choose the Build and Debug command in the Debug drop-down menu. Workshop will compile the Java code into a WAR file, deploy the application as a Web service, and display a browser-based debug harness. Have some fun with the debug harness. It will let you type in calling parameters for TWS and immediately see the results.

As described in the Java code of TWS, the `respond` method has two parameters. `wordcount` is an integer value that tells TWS how to compose the result. TWS uses the `Lingo.java` object to compose a String value containing a bunch of gibberish words. The `wordcount` value determines the quantity of gibberish words. Consider this example where Lingo returns a phrase with 50 gibberish words:

```
from com.pushtotest.tool.util import Lingo
myLingo = Lingo()
print myLingo.getMessage( 50 )

Ia quantos quantos delorum quantos. Delorum ditchek quantos
so quantos ad. Lipsem so novus surbatton deplorem deplorem.
```

```
Delorum surbatton novus delorum. Lipsem ditchek so. Lipsem
quantos it infreteres ad deplorem ipsem surbatton.
Ca nmpus it ad ipsem novus it surbatton delorum. Campus
ipsem quantos novus sit ditchek ventes aqua ad ad aqua
ditchek. Novato aqua deplorem it infreteres infreteres quantos.
```

The `respond` method will return a response containing Lingo text of the size determined in the request. The `delay` parameter determines the number of milliseconds to sleep before responding to the request.

Once you have accomplished this task, please move on to the next section, where you will configure and run the PerfCheck test agent to drive the running Web service.

PerfCheck

PerfCheck is an intelligent test agent written in TestMaker. Chapter 5 gives a thorough explanation of TestMaker. PerfCheck identifies the throughput of the TWS by making requests to TWS with increasing payload sizes and measuring the round-trip request/response time. PerfCheck averages the request/response times and presents a transactions-per-second result in the TestMaker output window.

PerfCheck issues requests to TWS for a predetermined amount of time, reports the TPS results metric, and then runs the test another time using a larger payload size. You should see the TPS value decrease as the payload size increases in accordance with the test results we found at PushToTest Labs and presented earlier in this chapter.

PerfCheck is organized among several script files to make the agent easier to understand. The following is a list and description of PerfCheck's contents.

- *properties.a*—Setup, configuration, and other parameters for the test.
- *master.a*—This is the main agent script to run. This script configures the test, stages the test, and records the results. The script displays the TPS and other performance metrics in the Output window.
- *tally.a*—Tally the results from the log file into TPS and other metrics.
- agents—A directory holding the scripts that implement the behavior needed to test the Web service.
- logs—A directory to hold the logged test result data.

Next, I take an in-depth look at these scripts and provide an explanation of how the test is constructed.

properties.a The *properties.a* file contains these parameters to control the payload size and duration of the tests.

```
timetorun = 8 * 60 * 1000
```

PerfCheck runs in cycles. Each cycle increases the payload size of the TWS responses. The duration of the cycle is set here. For example, 8 tells PerfCheck to run each payload size for 8 minutes.

```
agentcount = 50
```

At the beginning of each cycle, PerfCheck instantiates a quantity of test agent threads that run concurrently. Each thread makes a request to TWS, logs the response time and result code, and then makes another request. The `agent-count` parameter controls the total number of concurrent threads to run.

```
payloadstart = 500
payloadinc = 1000
paycount = 4
```

PerfCheck uses these parameters to determine the payload size to begin the test with. The `payloadinc` value increases the payload size for each cycle. The `paycount` variable determines the number of cycles PerfCheck will run.

```
tws_host = "http://examples.pushtotest.com"
tws_port = 7001
tws_path = "TWS/responder_doc.jws"
```

PerfCheck uses these parameters to identify the location of TWS. The BEA Weblogic Workshop debug harness will show you the URL to TWS. The debug harness will also show you the WSDL description of the TWS to identify the location of TWS.

master.a The *master.a* script stages the test by loading the property values, instantiating the test threads, and then running the *tally.a* script. To run a test, run the *master.a* script. The results will appear in the TestMaker output panel. A closer look at *master.a* shows how it accomplishes these tasks. Here is *master.a* in its entirety, followed by an in-depth explanation of the *master.a* script.

```
# master.a

from java.lang import Thread
import java
import time
from java.util import Date
from com.pushtotest.tool.logger import simplelogger
import sys
import thread

print "Configuring to run the test."

exec open( scriptpath + "Properties.a" ).read()

payloadsize = payloadstart

for c in range( paycount ):
    running = 1
    up_count = 0
    record = 0
    tick = 0
    errortick = 0

    first_time = Date().time

    logtarget = resultspath + "results_" + localname \
    + str(c) + ".txt"

    try:
        log = simplelogger.getInstance()
        log.setFileName( logtarget )
    except java.lang.Exception, ex:
        print "Could not start logging."
        sys.exit()

    total = 0
    for a in agents:
        total += a[1]

    if total>100:
        print "Warning: The total number of agents"
        print "         in the Properties.a file is"
        print "         more than 100%."

    for at in agents:
        agenttype = at[0]

        agent_count = int ( ( float( at[1] ) / \
        float( total ) ) * agentcount )

        for i in range( agent_count ):
            exec open( scriptpath + "agents/" + \
            agenttype + ".a" ).read()

            time.sleep( startupdelay )

    print "Threads started, we're testing!"

    record = 1

    start_time = Date().time
    end_time = start_time + timetorun

    while Date().time < end_time:
        time.sleep( 1 )

    record = 0

    recording_time = Date().time - start_time

    running = 0     # Time to stop

    log.closeLogFile()

    close_time = Date().time
    close_end_time = close_time + maxwait

    while ( Date().time < close_end_time ) and \
    ( up_count > 0 ):
        time.sleep( 1 )

    duration = Date().time - start_time

    exec open( scriptpath + "Tally.a" ).read()

    payloadsize += payloadinc
```

The *master.a* script is fairly straightforward and procedural. It starts the test, runs the test threads, then runs the *tally.a* script and cleans up the test.

Next I cover the *master.a* script step-by-step. Chapters 5, 7, and 11 feature other test agents that follow a similar design pattern as *master.a*.

```
from java.lang import Thread
import java
import time
from java.util import Date
from com.pushtotest.tool.logger import simplelogger
import sys
import thread
```

The *master.a* script begins with a few import statements. TestMaker provides the Jython scripting language. Jython is the popular Python programming language implemented 100% in Java. Details on Jython are in Chapter 5. The import statements identify the Python and Java objects Jython will use to perform the test. For example, later in this agent we will use the Thread object. This is Java's Thread object for running objects concurrently.

```
exec open( scriptpath + "Properties.a" ).read()
```

The `exec` command runs the script commands found in the *properties.a* script. The `scriptpath` variable is a TestMaker system variable that contains the path to the currently running script.

```
payloadsize = payloadstart

for c in range( paycount ):
```

master.a will cycle through tests with increasing payload sizes. The `payloadsize` variable controls the size of the payload. The `payloadsize` variable initializes to a value set from the *properties.a* script.

```
    running = 1
    up_count = 0
    record = 0
    tick = 0
    errortick = 0
```

These variables are used to control the concurrently running test threads we are about to create. `running` is a flag used to tell the test agents when to run and when to stop. `up_count` counts the number of agent threads that are up and running. `record` is a flag that tells the running test threads when to start logging results data. `tick` is used to keep track of the total number of completed transactions. `errortick` keeps track of transactions that ended in an error condition.

```
first_time = Date().time
```

`first_time` keeps track of the time the test started. This will be used later when the *tally.a* script reports the results.

```
logtarget = resultspath + "results_" + localname \
+ str(c) + ".txt"

try:
    log = simplelogger.getInstance()
    log.setFileName( logtarget )
except java.lang.Exception, ex:
    print "Could not start logging."
    sys.exit()
```

The *master.a* script uses TestMaker's log handler to output the test results to a log file. The test threads will use this log handler.

```
total = 0
for a in agents:
    total += a[1]

for at in agents:

    agenttype = at[0]

    agent_count = int ( ( float( at[1] ) / \
    float( total ) ) * agentcount )

    for i in range( agent_count ):
        exec open( scriptpath + "agents/" + \
        agenttype + ".a" ).read()
```

This code calculates how many of each type of thread to run based on the percentage value set in *properties.a*. For the purposes of this test the *properties.a* script uses only a single type of test thread. However, you may want to see if using a combination of SOAP encoding styles impacts scalability and performance in your environment.

The `agents` variable is a Jython list object that contains a set of tuples. A tuple here holds two values, the test thread name, and the percentage mix:

```
[ "wlw_agent_rpc_lit", 75 ]
[ "wlw_agent_docstyle", 25 ]
```

The first value of each tuple identifies by name the user archetype to use for a test agent and the second value indicates the percentage mix of the type of test agent to use when testing. For example, `["wlw_agent_rpc_lit", 75]` indicates that 75% of the total number of concurrently running agents will be using the `wlw_agent_rpc_lit` test thread behavior.

```
            time.sleep( startupdelay )
```

After each agent thread instantiation, the script uses the sleep method to give some time for the thread to bring up a client connection before starting the next one. Without this delay, the threads would start simultaneously and may overrun the environment's resources to handle instantiating all the threads. As a result, one or more threads might time out before connecting to the host.

```
    record = 1
```

All the threads are instantiated so changing the value of the record global variable tells the agents to begin logging results to the `log` object.

```
    start_time = Date().time
    end_time = start_time + timetorun

    while Date().time < end_time:
        time.sleep( 1 )
```

Wait around until the amount of time for the test elapses. While the *master.a* script sleeps, the instantiated test threads communicate with the target host.

```
    record = 0

    recording_time = Date().time - start_time

    running = 0     # Time to stop
    log.closeLogFile()

    close_time = Date().time
    close_end_time = close_time + maxwait
```

The test period is over. Setting the record global variable tells the running agents to stop logging results.

```
    while ( Date().time < close_end_time ) and \
    ( up_count > 0 ):
        time.sleep( 1 )
```

The *master.a* waits around while the test threads finish and exit. The *master.a* script then calls the *tally.a* script and exits.

test threads The test threads are defined in the agents directory. All of the test threads follow the same design pattern. Rather than showing the entire thread, I will show how each thread uses TestMaker to use the different SOAP encoding styles.

The *wlw_agent_rpc.a* script implements the code necessary to make SOAP RPC-encoded requests to the TWS. Here is a snippet of the script showing how to make the SOAP RPC request to the Web service:

```
protocol = ProtocolHandler.getProtocol("soap")
body = SOAPBody()
protocol.setBody(body)

protocol.setUrl( endpoint )

body.setTarget( "http://www.openuri.org/" )
body.setMethod( "respond" )

body.addParameter("wordcount", Integer, payloadsize, None)
body.addParameter("delay", Integer, 0, None )

response = protocol.connect()
```

The `body.addparameter()` method provides a powerful way to send objects to the Web service. The format allows you to add any object type that is on the TestMaker Java classpath. In this example we are sending two Integer data types. The first is the `wordcount` value and the second is the `delay` value.

Compare the relatively easy SOAP RPC encoded request to the next snippet of code. The following code is from the *wlw_agent_docstyle.a* script that implements the code necessary to make a SOAP document-literal encoded request to the Web service. As we saw earlier in this chapter, document-literal encoding receives an XML data structure already well formed and ready to send to the host.

```
protocol = ProtocolHandler.getProtocol("soap")
body = SOAPBody()
protocol.setBody(body)

protocol.setUrl( endpoint )

body.setMethod( "respond" )

ns = "http://www.openuri.org/"

elRespond = Element( "respond", ns )

ellen = Element( "wordcount", ns )
ellen.addContent( payloadsize + "" )
elRespond.addContent( ellen )

eldel = Element( "delay", ns )
eldel.addContent( "0" )
elRespond.addContent( eldel )

doc = Document( elRespond )

body.setDocument( doc )

response = protocol.connect()
```

This snippet of test thread script creates an XML document using the JDOM API that described in Chapter 7. The request document looks like this when built:

```
<s:Envelope
   xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <respond xmlns="urn:responder_msg">
      <delay xmlns="urn:responder_msg">25</delay>
      <length xmlns="urn:responder_msg">75</length>
    </respond>
  </s:Body>
</s:Envelope>
```

We use the JDOM APIs to build the XML request in a DOM object and then pass the DOM object to TOOL to make the SOAP call to the server. TestMaker uses the `namespace` parameter to identify the designation Web service name. The endpoint gets us to the right server, the namespace of the first element gets us to the right Web service.

Finally, the last snippet of code shows the *wlw_agent_rpc_lit.a* scripts use of SOAP RPC-literal encoding. In this snippet we use the same JDOM APIs to construct the XML document. However, we will use the SOAP RPC encoding commands to pass the XML document as a single element.

```
protocol = ProtocolHandler.getProtocol("soap")
body = SOAPBody()
protocol.setBody(body)

protocol.setUrl( endpoint )

protocol.setEncoding( Constants.NS_URI_LITERAL_XML )

body.setTarget( "http://tempuri.org/responder_doc" )
body.setMethod( "respond" )

elOne = Element( "respond" )

elDelay = Element( "delay" )
elDelay.addContent( "12" )
elOne.addContent( elDelay )

elLength = Element( "length" )
elLength.addContent( "25" )
elOne.addContent( elLength )

domo = DOMOutputter()

body.addParameter( "inDoc", w3c_element, \
domo.output( elOne ), Constants.NS_URI_LITERAL_XML )
```

This code snippet is similar to the previous snippet that used SOAP document-literal encoding. This code used the SOAP RPC encoding calls with the following difference: `setEncoding( Constants.NS_URI_LITERAL_XML)` tells TestMaker that the parameters of the request and response will use document-literal encoding. Additionally, the `addParameter` method passed in a `w3c_element` type of object containing the XML data.

As you can see the kit provides a flexible and powerful way to test the popular SOAP encoding styles for scalability and performance. To start a test, open TestMaker, navigate to the kit test agent scripts, and open and run the *master.a* script. The script displays the TPS and other performance metrics in the Output window.

Some Other Things We Found

While building and running the test environment for Elsevier, I noticed a few things worthy of mentioning here. First, each application server platform I worked with has its own implementation of a SOAP stack. Some even shipped with more than one stack. For example, BEA WebLogic server comes with a SOAP stack that implements the JAX-RPC API and one that implements the JWS APIs. IBM WebSphere comes with the DOM-based Apache SOAP library, but also has the SAX-based Apache AXIS library. I would love the opportunity to test the differences between all of these in some future project.

I found that while WSDL did a fair job at describing the interface to a SOAP service, many times WSDL was not complete. I needed to use a network monitor to see actually what values were moved over the HTTP transport.

I also noted that while building the test environment for Elsevier, BEA and IBM announced major new versions of their Web service application servers. Each new version changed their SOAP stack implementation with positive and negative impact on performance and scalability.

Reliability and performance of SOAP application servers is a moving target. This further reinforced the lesson that a test environment is necessary to check reliability and performance now and as new implementations become available.

With the test environment complete, Elsevier Science now has the means to understand reliability, scalability, and functionality of its systems. The test agent technology will be employed as new builds of the Elsevier software and the Web service application servers become available. The test environment is planned also to provide ongoing tests for regression, functional tests, and as a proof-of-service monitor.

Summary

In this chapter, we found that software developers have many choices for building Web service systems: SOAP encoding styles, Web service development tools, and application servers. This chapter presented the results of an investigation that shows how each choice immediately impacts scalability and reliability.

Elsevier adopted SOAP as their standard way to build their next-generation content aggregation system. We saw how Elsevier developed a new methodology and test environment to check various SOAP implementations, including application servers, SOAP stacks, and utilities, for scalability and performance.

15 Multiprotocol Testing in an Email Environment

Saint Leo University (http://www.saintleo.edu/) is the third largest provider of online higher education in the world and the sixth leading provider of higher education to the U.S. military. The IT staff found TestMaker by searching for open source test tools that check email systems for scalability and performance under load. In a few months their students, faculty, and administrators will return for a new session and begin using a new email system. The question they are asking themselves is: "Did we buy the right email software package, network bandwidth, and servers?"

This chapter presents the results of an investigation to answer this question. It discusses the needs of the university to conduct a multiple protocol test in an email environment, the choices of test tools and methodologies, describes in-depth the test agent code used to conduct the test, and shows how the university analyzed the results.

Needs and Benefits

Modern universities depend on email for everything. The university information system serves students when they are on campus, students living off campus, and students that attend sessions at satellite campuses who will rarely be on campus. Plus, the faculty uses the same email system to coordinate teaching assignments, provide assignments to students, and communicate with administrators and staff.

IT managers are in a difficult balancing act. The goal of maintaining an email system is to be efficient on bandwidth and server and router equipment without pressing the system into boundary conditions. In a boundary condition, hard drives run out of space, users cannot access the system because of overloaded network connections, and system administrators have no insight into these problems because the system is too overburdened to even produce administrative reports. It is up to the IT manager to balance the needs for efficiency with the requirement for availability and system readiness to serve users.

Testing the email system provides two benefits to the university. It uncovers problems in the current configuration before students, faculty, and staff encounter the problems. And by providing a better understanding of how the email system handles the load of its users, the university will save money by avoiding unnecessary server and router purchases and by negotiating better contracts with its Internet service providers.

Like any institution, budgets are driven by the need for efficiency at the university. So, the IT staff was looking for an open source tool simply to save their budget from commercial software test tool licensing costs. They were attracted to TestMaker because of its multiprotocol support. A TestMaker test agent drives an email service, as a real user would use the email service's native protocols. For example, a student using Microsoft Outlook Express drives the email system using SMTP, POP3, and IMAP protocols. At the same time, a student checks email messages with a browser using HTTP protocols. TestMaker test agents use native HTTP, SMTP, POP3, and IMAP protocols just as students will.

The IT staff could have used TestMaker to build its own test agents; however, they were time constrained to conduct the tests within a few weeks and some of its engineers were enjoying the break between sessions. Saint Leo University asked PushToTest to provide it with consulting services to write the test and to conduct a scalability investigation of the new email system.

Scoping the Project

Saint Leo University provides online university courses, so renovations of its online systems are a regular part of its operations. In its most extensive renovation yet, the university changed its underlying infrastructure to provide more of its content to students and faculty through an enhanced Web portal. The university needs to integrate its multiple data systems and technologies at its many locations into one seamless online system, to manage its online applications and forms, and to develop online discussion groups for alumni and students. The project included upgrading its email capability. For this, the university chose Stalker Software's CommuniGate Pro messaging server.

CommuniGate Pro is an established messaging server among university campuses. The software provides a Web interface and standard SMTP/POP3/IMAP interfaces for users to check and send email messages. The software installs on inexpensive Intel-based servers and provides low-maintenance administration to keep operating. While the software is well designed and engineered, the needs for network bandwidth, server, and routing equipment to run an efficient email system are less well known. The university provides 12,000 users with email accounts each semester. With the rollout of their new email system immanent, the university needs to know if their system, datacenter, network bandwidth, and software configuration are ready.

PushToTest began the project by taking a survey, which is in Table 15–1.

Table 15–1 A Survey on the Scope of the Email Test Project

Question	Answer from IT staff
What is the maximum number of concurrent users that will be on the email system at any given time?	1,000
When do you expect peak usage, and for what reason?	Peak usage would happen between 8:00 a.m. and 10:30 a.m. Mon–Fri. Reason: Students/employees retrieving emails.
What is the speed and nature of the Internet connections that your users will use to connect to the email system?	We currently have a 3 MB pipe to the Internet thru Sprint. (This will be upgraded in the near future.) Users will connect on-campus and off-campus to the email system using Outlook, Netscape, and a WEB > interface.
Will you provide IMAP, SMTP, and POP3 service? Any others?	IMAP, POP3, SMTP, as well as a WEB interface using IMAP. Also we will be using the CommuniGate MAPI protocol to connect select (100–200) staff and faculty to calendar services on the server.

Table 15–1 A Survey on the Scope of the Email Test Project (continued)

Are you using antispam software utilities in your email system? If so, what are they?	We are using Real Time Blacklists.
Are you using antivirus software utilities in your email system? If so, what are they?	Yes. Sophos.
If you were to describe three types of users (archetypes or prototypical users) of the email system, what would they be and how would they use the email system?	User 1. Off-site student. Uses SLU mail for communication sent to and from SLU staff. User 2. On-site student. Uses SLU mail for all communication to and from SLU staff and family/friends. User 3. Faculty/staff member. Uses SLU mail for all communication to students and fellow staff. Large attachments from MS Office.
What administrative policies will govern each email account? For example, maximum storage allotment, maximum number of email messages.	Max storage of messages for a student is 20 MB. Staff/faculty limits are at 40 MB. Of course special exception is granted for many users. We plan on limiting the special exception cases to 200 MB of storage. Our max number of emails isn't set. We have limits with the Sophos plug-in on # of messages (5,000) sent/received in one hour.
How will you provision new user accounts?	Storage limited to 20 MB for student and 30 MB for staff. Student accounts are run into the system in batch mode twice per week. Faculty/staff must get sign off from HR Dept. before they are added.
How many technicians do you have to operate the email system? What are their typical skills and experience?	We have two help desk techs to set up accounts. We have one FTE position for maintenance and administration of the server.

The survey and answers told me that testing the university email system is an appropriate use for TestMaker. I also spoke with Stalker Software to learn their input on the test. They gave a positive opinion and also gave advice on the test scope of the project and the test goals. As you will see in the next section, the survey also gave a good start at defining the test itself.

Test Design and Methodology

The survey presented in the previous section of this chapter gives a lot of insight into how to design the test and test methodology. The test goal is to learn if their existing system configuration, network bandwidth, and server configuration satisfactorily serves the university user community. PushToTest designed the test to emulate a group of virtual users to load the system. The virtual users record the time it takes to complete each transaction. The test then analyzes the results to determine the throughput of the system under load, measured in TPS. Additionally, while the test is running, the IT staff will see firsthand the test's impact on system resources (disks, CPU bandwidth, network IO) and network bandwidth.

The test design includes the resources illustrated in Figure 15–1.

In a distributed test environment, test agents are run on multiple test server devices. Each server, called a TestNode and running the PushToTest TestNetwork software, generates load on the email server by emulating the actions of archetypal users. To the email server, it appears that the requests coming from the TestNodes are from real users. Details on the user archetype design are found later in this section.

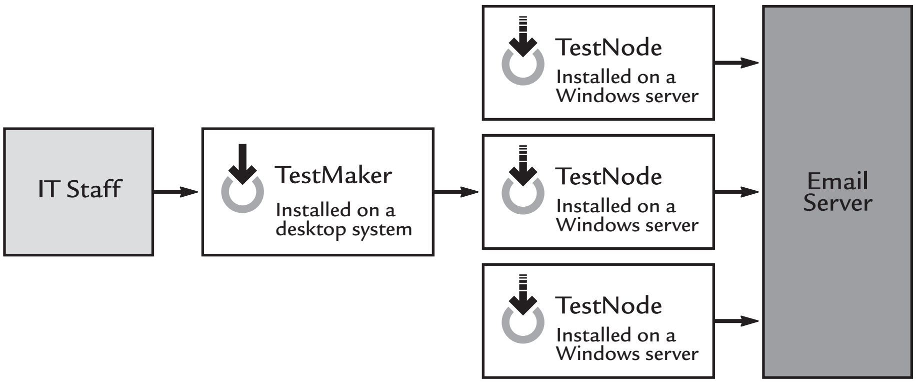

IT Staff
TestMaker
Installed on a desktop system
TestNode
Installed on a Windows server
TestNode
Installed on a Windows server
TestNode
Installed on a Windows server
Email Server

Figure 15–1 The email system test uses TestMaker as a central console to drive the test agents running in several TestNodes. The TestNodes for this test are inexpensive Intel-based Windows servers running the TestNetwork software. The test agents emulate virtual users to drive load on the email server.

The distributed test environment provides a lot of flexibility to the university IT staff. They may run the TestNodes in a lab setting to load test the email server. They may also distribute the TestNodes across their network infrastructure to learn which parts of their network are underperforming. They may also put the TestNodes on different Internet connections to the email server. For example, the university provides connectivity to the email server through connections provided by Time Warner, Verizon, and others. By putting a TestNode on each external network, the university learns the latency differential between the provided networks.

TestMaker provides a single console to stage and synchronize a test of the email system. TestMaker comes with client software to control TestNodes. The TestMaker command protocol uses SOAP messages over HTTP connections to send instructions to the TestNodes. This design means the university did not have to change its configuration, including opening new ports on its firewall, to use the distributed test environment inside and outside of the university network.

TestMaker provides a graphical user interface with which to start a test, monitor the test's progress, and analyze and present the results. The university installed TestMaker onto an IT team member's desktop computer and controlled the TestNodes from there. TestMaker and TestNetwork are 100% Java applications and run wherever Java runs. So when the IT staff roles change, the environment may easily be moved from Windows to Linux or Mac OS X, or anything else that runs Java.

The test methodology follows the steps listed in Table 15–2.

Table 15–2 The Email System Test Methodology

Step	Goal	Action and measurement
1	Configure the system for a test.	Email system installed and ready to serve users. 500 test user accounts created. TestNetwork software installed on TestNode machines. TestMaker software installed on IT staff desktop computer.
2	Identify the capacity of the TestNode machines to run test agents.	Run a ramp-up test script that runs increasing levels of intelligent test agents on the TestNode machine. Each agent emulates one virtual user. The script analyzes the agent performance to determine the highest number of concurrent test agents to run before the local machine runs out of CPU, memory, and network bandwidth.

Table 15–2 The Email System Test Methodology

3	Write intelligent test agents for each user archetype.	The survey uncovered three archetypal users of the email system: on-campus students, off-campus students, and faculty. Three test agents implement the behavior of each three archetypal users. A description of the user archetypes comes later in this chapter.
4	Conduct a trial run of the test system.	Run a test for 5 minutes to determine that the steps taken in 1–3 of the test methodology are configured correctly.
5	Run the test.	Run successively larger sets of virtual users against the email system—500, 1,000, 1,500, 2,000, 2,500, and 3,000—across all the TestNodes. Run each test for 10 minutes.
6	Run the Tally script.	The Tally script analyzes the test result logs to determine the transactions-per-second index and other statistics.
7	Repeat steps 5 and 6, 3 times	A transactions-per-second index varies more than 10% from the tests run in Step 5 indicates an external force is at work that impacts the test results.
8	Run the test.	Run the test for 1 hour.
9	Run the Tally script.	Contrast the results statistics from the short and long test runs.
10	Search for anomalies in the test and in the system.	Construct a list of system properties that may impact the test results.
11	Document the test and findings.	Include the test methodology, test statistics, and observations.

User Archetypes

Intelligent test agent technology is a powerful new way to test a system according to the goals of archetypal users. Rather than testing raw functionality in the university email system, a test that models itself on user archetypes shows how close the system brings users to their goals. Chapter 2 provides detailed discussion on how to define a user archetype in the context of a test for scalability, reliability, and performance. In the case of the university email system, the survey uncovered three user archetypes. With some further investigation, PushToTest was able to construct the following user archetypes:

- Marilyn is an off-site student that uses the email system for communication sent to and from the university faculty. Marilyn is an undergraduate student who has a full-time job in a legal office as a clerk. Marilyn uses the email Web interface to check for messages. When new email is waiting, she immediately reads and deletes the email messages. Marilyn sends an email message to a friend every time she checks for messages.
- Muriel is an on-site student that uses the email system for communication sent to and from faculty members advising her. Muriel is a senior and has a part-time job as a teacher's aide. Muriel uses a Windows-based laptop with Microsoft Outlook Express. Muriel is impatient and proactive by nature. When new mail is waiting, she immediately downloads and reads the email. Muriel sends `reply` messages to any email she receives from the faculty right away.
- Betsy is a tenured professor who uses SLU email for communication with students and fellow staff. Betsy uses the SLU SMTP/POP3 interface to check for email. Betsy likes to handle email messages in groups. She normally waits for five messages to appear before reading all of the messages in her mailbox. Betsy puts her students into groups of 10 to support each other. From time to time she sends messages with Microsoft Office attachments (Excel, Word, and Zip format) to each of the groups to make sure they have the latest assignment.

These user archetypes demonstrate the behavior of three prototypical users of the university email system. They cover students that are on campus, students that are off campus, and students that attend sessions at satellite campuses who will rarely be on campus. Plus, the Betsy archetype has the behavior of a faculty member using the email system to coordinate teaching assignments, provide assignments to students, and communicate with administrators and staff. Of course, there is room for many more archetypes to test the system, including archetypes for the IT staff, for example. Later, this chapter presents the user agents that are modeled after these user archetypes.

Installation and Configuration

The test environment installation requires these steps to be taken by an IT staff member at the university:

- Install TestMaker onto a desktop computer system. TestMaker comes as a ZIP archive file. Unzipping the files creates a TestMaker directory containing all of the files, scripts, utilities, and libraries to run intelligent test agents. TestMaker requires Java 1.4 and higher.
- Install TestNetwork onto the TestNode machines. TestNetwork comes as a ZIP archive file. Just like TestMaker, TestNetwork comes with all files, scripts, utilities, and libraries to operate a TestNode. By default, TestNetwork configures itself as a SOAP-based Web service that is accessed on port 8090. The TestNetwork installation instructions show how to configure for a different port number. TestNetwork requires Java 1.4.
- Install the test agent scripts onto the machine running TestMaker. The test agent's scripts include files described in Table 15–3.

Table 15–3 Test Agent Files

File name	Description
EmailTestScenario	Contains all the files that will be sent to TestNodes to stage the email test, including the following: *properties.a*—Configuration and settings for the test. For example, controls the relative mix of test agents based on user archetypal behavior. *master.a*—The main entries point to setup, run, and take down the test. *archetypes*—Directory containing scripts that implement the user archetype behaviors by using TestMaker's TOOL library to drive the email system using native SMTP, POP3, IMAP, and Web protocols.
nodes.conf	A properties file that defines the TestNodes to be used in a test.
testmail.a	The main entry point for the entire test. This script stages the test agents to the TestNodes, passes all needed parameters, and returns the resulting logged result data to the TestMaker machine.

- Configure the email system to have a set of email accounts for the test agents to use.
- Set the *properties.a* file to configure the intended mix of test agent archetypes, the URLs to the POP3 and SMTP servers, the URL to the Web host, and the quantity of virtual users.

With these installation and configuration steps, the test is ready to be implemented. Before we delve into the results from the test, the next section of this chapter shows how I went from the user archetype descriptions to the test agent scripts.

From Archetypes to Test Agent Scripts

User archetypes describe a prototypical user of a system. Each system will have many archetypes, but each archetype is the full description of a certain type of behavior. For example, my Uncle Norman is the archetype of a 1960s bachelor. He is an American that grew up in New York City and moved to Rome to be an airline pilot. He has no children, finally got married at age 64, and has an incredible talent to maintain friendships around the world. His lifestyle, manners, sense of humor, and tastes are those of a 1960s bachelor. While there are hundreds of thousands of people from a similar background, after describing him to you, I do not need to describe anyone else from that era or social background. The same is true about a user archetype. Once described, the archetype describes the unique behavior of a single person but represents a group of people.

The survey presented earlier in this chapter identified three user archetypes that we identified as Marilyn, Muriel, and Betsy. Eventually we will add many more user archetypes that will increase the breadth of the test to encompass more user goal-oriented tests. For each new archetype we will ask ourselves, is the new archetype really one of the existing ones? For example, suppose we develop a new archetype named Simon who is a freshman student living in the campus dorms who turns in his assignments as email attachments to his teachers. Unless there is more to Simon, I would put it to you that Simon is really Muriel with the ability to send email attachments. Rather than invent a new user archetype, we could expand Muriel to include a behavior where Muriel sends files to her teachers. User archetypes represent prototypical behavior that is far from me-too behavior.

Moving from user archetypes to test agents that implement the archetype's behavior requires some effort in the TestMaker environment. Chapters 4 and

5 provide a good description of the lifecycle of an intelligent test agent, from creation to retirement. Applying those techniques to the university email system test is fairly straightforward.

Building Muriel

Earlier in this chapter we identified Muriel as a student that uses the email system while on the university campus. Muriel uses the email system to communicate with her faculty advisors. She is a senior and has shown herself to be a terrific asset to the teaching staff by being a part-time teacher's aide. She rapidly checks for new mail messages and immediately downloads and replies to email messages when they arrive. Muriel is especially vigilant to reply to messages she receives from the teaching staff.

Translating the Muriel archetype into intelligent test agents needs to accomplish these four behaviors:

- Muriel uses an email client that checks for new email using SMTP and POP3 protocols.
- When new mail is waiting, she immediately downloads and reads the email.
- Muriel sends reply messages to any email she receives from the SLU staff. For the purpose of this test, Muriel considers mail received from the test1 to test50 accounts to be from a faculty member.
- Muriel sends email messages with no attachments at random intervals from 30 seconds to 60 seconds to other students.

The following code implements the basic logic of an intelligent test agent modeled after Muriel. Here is the relevant snippet of test agent script, followed by an explanation.

```
while running:

        status = "ok"
        datapulse = Date().time

        try:
            read_emails( agent_num )

            chattercount += 1
            if chattercount > random.randrange( 1, 3 ):
                chattercount = 0
                send_email( agent_num, \
                  random.randrange( 51, 100 ) )
```

This code snippet says that as long as the test is running, the test agent will read email messages from the test account identified to this test agent. Randomly, the test agent will compose and send an email message to one of the students.

When the agent reads an email message, if the message comes from a faculty member, meaning it comes from test1 to test50, then the agent will send an immediate reply. To implement this test agent logic, the test agent includes a function to read email messages. Here is the function in its entirety, followed by a detailed description of the moving parts of this agent.

```
def read_emails( agentnum ):

    global mailhost

    replyaccount = 0
    fromadr = ""

    protocol2 = ProtocolHandler.getProtocol("mail")
    protocol2.setHost( mailhost )

    account = "test" + str( agentnum )
    protocol2.setUsername( account )
    protocol2.setPassword( "passw0rd" )

    response = protocol2.connect()

    response.setPermission( Folder.READ_WRITE )
    response.setFolder( "INBOX" )
    messages = response.getMessages()
    replyaccount = 0

    for i in messages:
        if ( i.getFrom() != None ):
            fromadr = str( i.getFrom()[0] )
            replyaccount = 0

            if fromadr.lower().find("test") == 0:
                pos = fromadr.lower().find("@")
                if pos != -1:
                    account = fromadr[ 4 : pos ]

                    if int( account ) < 50:
                        replyaccount = int( account )

            i.setFlag( Flags.Flag.DELETED, 1 )

        response.close( 1 )

        if replyaccount > 0:
            send_email( agentnum, replyaccount )
```

The test agent implements Muriel's need to reply to email messages that come from the faculty. Next I show how this function works in detail.

```
def read_emails( agentnum ):

    global mailhost

    replyaccount = 0
    fromadr = ""
```

First we define the name of the function as `read_emails` that takes a single parameter. The `agentnum` is an integer value of the agent thread number. The agent number forms the email account name to check or send messages. For example, when `agentnum` equals 8, then `read_emails` will check the test8@universitymail.com mail account.

```
    protocol2 = ProtocolHandler.getProtocol("mail")
    protocol2.setHost( mailhost )

    account = "test" + str( agentnum )
    protocol2.setUsername( account )
    protocol2.setPassword( "passw0rd" )
```

The test agent script creates a new instance of the `MailProtocol` object that we will reference in the `protocol2` variable. Chapter 8 provides details on the `MailProtocol`. The additional commands here tell the `MailProtocol` object where to find the email host and which account to use.

```
    response = protocol2.connect()
```

The `connect()` method opens a connection to the mail host, authenticates the test agent, and returns a `response` object that is used for subsequent email operations.

```
response.setPermission( Folder.READ_WRITE )
```

The `setPermission` method instructs the mail host on our test agent's intentions. By default, `Folder.READ_ONLY` tells the mail host we will be reading email messages but not deleting them from the host. `Folder.READ_WRITE` tells the mail host to expect message delete commands.

```
response.setFolder( "INBOX" )
```

The POP3 protocol uses a single folder titled INBOX by default. The IMAP protocol supports multiple folders.

```
messages = response.getMessages()


for i in messages:
    if ( i.getFrom() != None ):
        fromadr = str( i.getFrom()[0] )
```

Iterate through each message. The `getFrom`, `getSubject`, and `writeTo` methods return the from, subject, and message body.

```
        replyaccount = 0

        if fromadr.lower().find("test") == 0:
            pos = fromadr.lower().find("@")
            if pos != -1:
                account = fromadr[ 4 : pos ]
```

This test agent script decodes the value of the *to* address for the email message. The test agent users accounts follow the same format, test1@universitymail.com, test2@universitymail.com, etc. This code finds the account number from the email from address.

```
                if int( account ) <= 50:
                    replyaccount = int( account )
                    send_email(agentnum,replyaccount)
```

Muriel replies to email messages she receives from the faculty immediately. For the purpose of this test, email accounts less than test50 are considered to be faculty members. When the email message is from a faculty member (account number ≤ 50), then Muriel sends a reply message. `Send_email` is another defined function that takes two parameters, the current agent number and a destination email account number, and sends an email message.

One extra step we could take in this test would be for Muriel to read the contents of the message to validate the communication. The body of the message is available from the `writeTo` method of the `response` object, for example, print `i.writeTo(outputStream)`.

```
        i.setFlag( Flags.Flag.DELETED, 1 )
```

The `setFlag` method enables the agent script to mark the email message for deletion from the mail host. The flag is set in this loop but does not actually happen until the `close()` method executes. The flag has no effect if the `setPermission` method is not set to `Folder.READ_WRITE`.

```
    response.close( 1 )
```

Closes the connection to the host. The `close` method takes a single Boolean parameter. When the parameter is set to true or 1, then the `Close` command instructs the mail host to delete any flagged messages. A `False` or `0` parameter tells the mail host to ignore the deleted messages.

This test agent script implements Muriel's behavior. The Muriel test agent considers mail received from the test1 to test50 accounts to be from faculty members. The test agent replies to email messages received from these accounts. The test agent regularly sends email messages to other students, those with accounts greater than test51. Lastly, the test agent simulates using Microsoft Outlook Express by using native SMTP and POP3 protocols as Outlook would to send and read email messages.

Next, we see how to build an intelligent test agent modeled after Marilyn.

Building Marilyn

As we saw earlier in this chapter, Marilyn is an off-site student that uses the email system for communication sent to and from the university faculty. Marilyn uses a Web browser to read and send email messages. This gives us the opportunity to use TestMaker's Recorder wizard to help write the test agent script.

Using the Recorder

The Recorder is a useful and convenient utility function in TestMaker that watches you use a browser and then it writes a test agent script for you automatically. Marilyn uses a Web browser to send and receive email messages. We used the Recorder to record Marilyn's actions reading and sending email messages. Then we modified the recorded script to be part of the overall test. Figure 15–2 illustrates starting the Recorder in TestMaker.

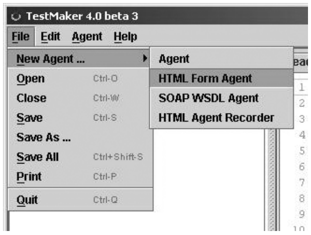
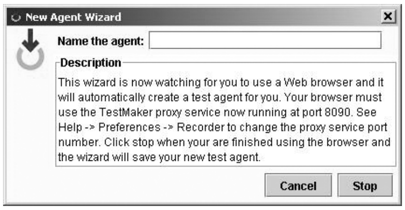


Figure 15–2 Start the Recorder in TestMaker by choosing the New Agent command in the File drop-down menu, then choosing HTML Agent Recorder.

Activating the Recorder displays a dialog shown in Figure 15–3. The dialog asks you to name the recorded test agent script.


New Agent Wizard

Name the agent:

Description

This wizard is now watching for you to use a Web browser and it will automatically create a test agent for you. Your browser must use the TestMaker proxy service now running at port 8090. See Help -> Preferences -> Recorder to change the proxy service port number. Click stop when your are finished using the browser and the wizard will save your new test agent.

Figure 15–3 The Recorder starts by asking for the name of the recorded test agent.

TestMaker implements the Recorder as a special type of proxy server. Your browser makes requests through the Recorder proxy server. The proxy passes through the browser requests to the Web host, plus it records the request as a series of test agent script commands.

Muriel begins each session by identifying herself to the email system. Figure 15–4 shows the Web Mail sign-in page.

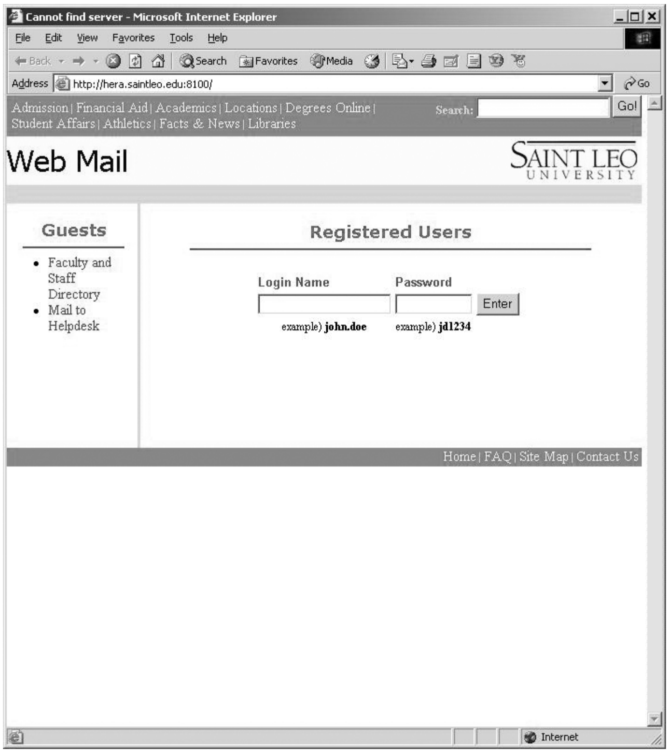

Cannot find server - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://hera.saintleo.edu:8100/
Go
Admission | Financial Aid | Academics | Locations | Degrees Online | Student Affairs | Athletics | Facts & News | Libraries
Search: Go!
Web Mail
SAINT LEO UNIVERSITY
Guests
Faculty and Staff Directory
Mail to Helpdesk
Registered Users
Login Name
Password
Enter
example) john.doe
example) jd1234
Home | FAQ | Site Map | Contact Us
Internet

Figure 15–4 All sessions on the university web mail system starts with the sign-in page. Here the user enters their login name and password. All subsequent pages identify the user with a session identifier in the URL of each page.

Behind the scenes, the Recorder watches the sign-in page and writes the following test agent script. First I will show you the snippet of test agent script and then I will break it down and provide a detailed explanation.

```
loc = '''http://hera.saintleo.edu:8100/'''

http = ProtocolHandler.getProtocol("http")
http.setUrl( loc )
http.setType( HTTPProtocol.POST )
body = ProtocolHandler.getBody( 'http' )
http.setBody( body )

body.addParameter('''Username''', '''test''' + \
str( agentnum) )
body.addParameter('''Password''', '''apassword''')
```

```
    body.addParameter('''login''', '''Enter''')

    response = http.connect()
    if response.getResponseCode()<>301:
        raise Exception, "Sign-in response code: " + \
        str( response.getResponseCode() )

    # Follow redirect to mailbox menu page

    loc = response.getParameterValue( "Location" )
    pos = loc.find("Session")
    pos2 = loc.find( "/", pos + 9 )
    sessionid = loc[ pos + 8 : pos2 ]

    http.setUrl( loc )
    http.setType( HTTPProtocol.GET )
    response = http.connect()
    if response.getResponseCode()<>200:
        raise Exception, "Follow response code: " + \
        str( response.getResponseCode() )
```

This snippet of test agent script performs the sign-in function that Marilyn operates using a Web browser. Provided the user id and password are correct, the sign-in page servlet responds with a redirect command to ask the browser to load the Hello.wssp page. The following is a detailed explanation of this test agent script snippet.

```
    loc = '''http://hera.saintleo.edu:8100/'''
```

The new `loc` variable will contain a String value of the URL to the sign-in page.

```
    http = ProtocolHandler.getProtocol("http")
    http.setUrl( loc )
    http.setType( HTTPProtocol.POST )
    body = ProtocolHandler.getBody( 'http' )
    http.setBody( body )
```

The test agent uses HTTP protocols to command the email Web interface. These commands create an HTTP protocol handler object that will issue a `POST` command to the host.

```
    body.addParameter('''Username''', '''test''' + \
    str( agentnum) )
```

```
body.addParameter('''Password''', '''apassword''')
body.addParameter('''login''', '''Enter''')
```

The body of the HTTP protocol carries the user id and password for Marilyn.

```
response = http.connect()
if response.getResponseCode()<>301:
    raise Exception, "Sign-in response code: " + \
    str( response.getResponseCode() )
```

The `connect` method issues the `POST` command to the email host and creates a `response` object. The script checks to make sure the email host returns a 301 Redirect command to the Hello.wssp page. If the sign-in function did not perform as expected, the script raises an exception and returns.

Special note needs to be given to the response of the sign-in page. The response includes three pieces of data:

1. Response code—this is going to be a 301 Redirect code.
2. URL to the next page—this is either going to be the URL to the Hello.wssp page or to an error page.
3. Session identifier—the university email system is configured to use URL rewriting to identify a user to the host. Every subsequent request to the Web Mail host must contain the session identifier in the URL of the request. Otherwise, the Web Mail host will redirect the request to the sign-in page. URL rewriting is an alternative to browser cookies. Rather than keeping the session identifier in a cookie, the Web Mail host keeps the session identifier in the URL. For example, here is a Web Mail URL for a user that is signed in:

```
http://hera.saintleo.edu:8100/Session/14665-  \
y9jgEy1H7IKcrDJxcoAo/Hello.wssp
```

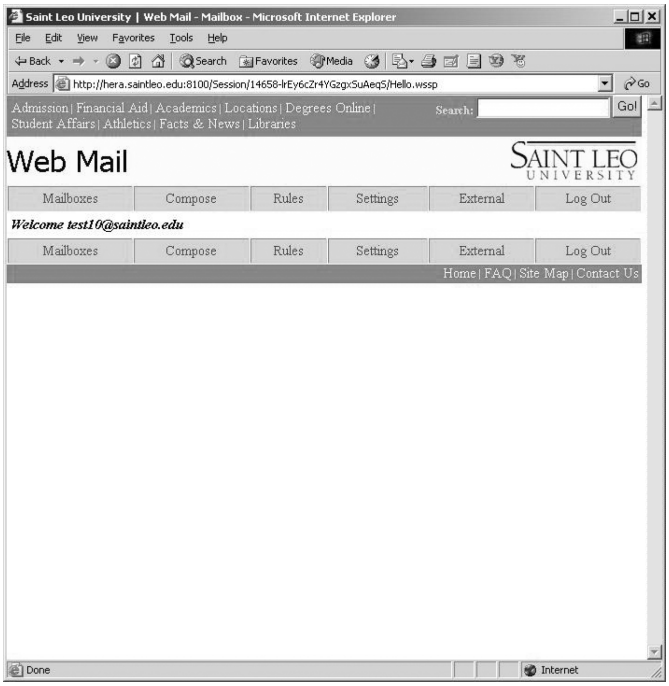
Unfortunately, the Recorder does not support dynamic URL rewriting during playback. If we used the recorded script, then the test agent would sign in and then use the recorded session identifier. Usually systems are configured to expire a session identifier after a certain amount of time. So, the recorded session identifier would be invalid. To avoid these problems, I added the following script to the recorded test agent script:

```
    loc = response.getParameterValue( "Location" )
    pos = loc.find("Session")
    pos2 = loc.find( "/", pos + 9 )
    sessionid = loc[ pos + 8 : pos2 ]
```

This code dynamically finds the session identifier in the redirect URL. It stores and uses the session identifier value for subsequent requests to the host.

```
    http.setUrl( loc )
    http.setType( HTTPProtocol.GET )
    response = http.connect()
    if response.getResponseCode()<>200:
        raise Exception, "Follow response code: " + \
        str( response.getResponseCode() )
```

This test agent script uses the same HTTP protocol handler object to get the redirected page—the mailbox menu page. A response code of 200 indicates success. In the browser, we see the interface shown in Figure 15–5.


Saint Leo University | Web Mail - Mailbox - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Go
Admission | Financial Aid | Academics | Locations | Degrees Online | Student Affairs | Athletics | Facts & News | Libraries
Search: Go!
Web Mail
SAINT LEO UNIVERSITY
Mailboxes Compose Rules Settings External Log Out
Home | FAQ | Site Map | Contact Us
Done
Internet

Address http://hera.saintleo.edu:8100/Session/14658-lrEy6cZr4YGzgxSuAeqS/Hello.wssp

Welcome test10@saintleo.edu

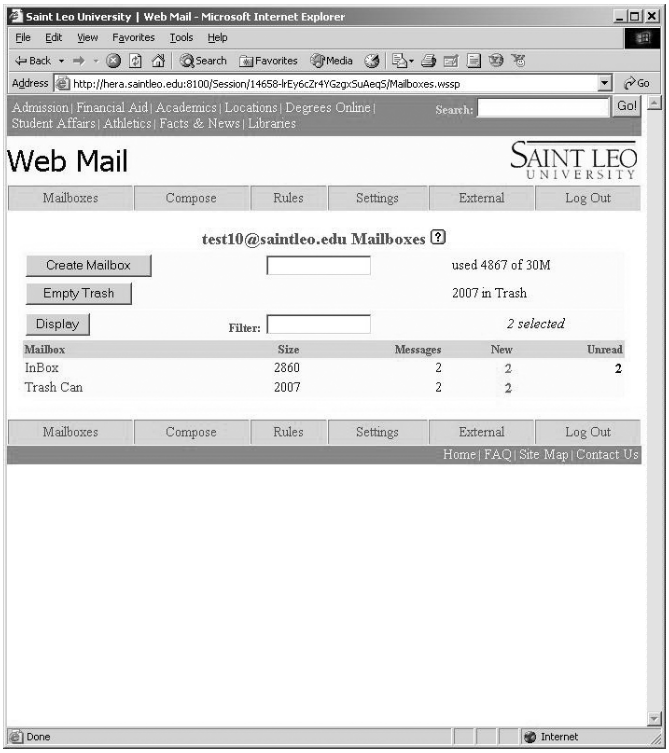
Figure 15–5 After the sign-in page, the web mail interface displays a page that lets the user choose to view their mailbox, compose new mail, and choose a number of other options. Marilyn goes right to the mailbox to read any waiting messages.

Marilyn wants to get to her mailbox so she clicks the Mailboxes link. The test agent uses the following script commands to get the Mailboxes.wssp page.

```
loc = "http://" + mailweb + "/Session/" + sessionid\
+ "/Mailboxes.wssp"

http.setUrl( loc )
http.setType( HTTPProtocol.GET )
response = http.connect()
if response.getResponseCode()<>200:
    raise Exception, "Mailbox summary response code: "\
    + str( response.getResponseCode() )
```

Clicking on the Mailboxes link takes us to the Mailboxes.wssp page, as shown in Figure 15–6. This page summarizes the contents of the mailboxes.


Saint Leo University | Web Mail - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address
Go
Admission | Financial Aid | Academics | Locations | Degrees Online | Student Affairs | Athletics | Facts & News | Libraries
Search: Go!
Web Mail
SAINT LEO UNIVERSITY
Mailboxes Compose Rules Settings External Log Out
Create Mailbox
Empty Trash
Display
Filter:
Home | FAQ | Site Map | Contact Us
Done
Internet

http://hera.saintleo.edu:8100/Session/14658-lrEy6cZr4YGzgxSuAeqS/Mailboxes.wssp

test10@saintleo.edu Mailboxes

used 4867 of 30M

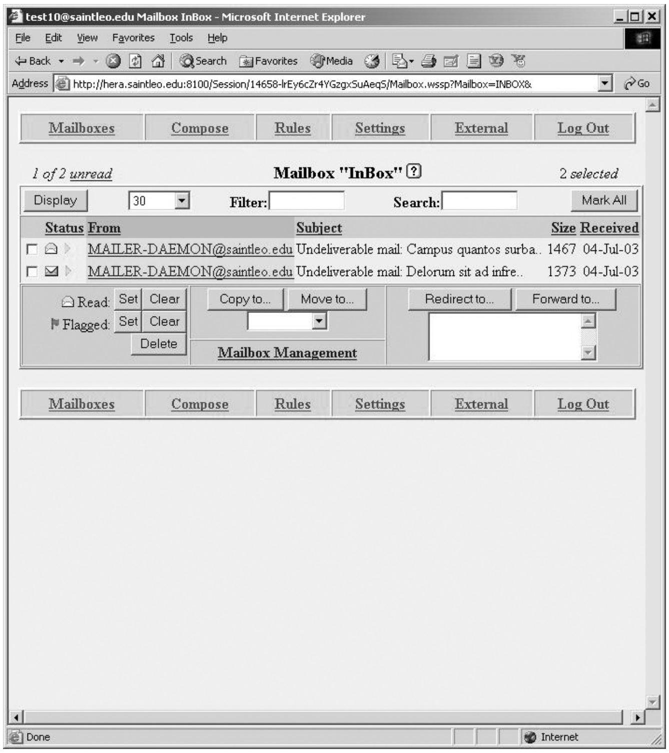
2007 in Trash

2 selected

Mailbox	Size	Messages	New	Unread
InBox	2860	2	2	2
Trash Can	2007	2	2	

Figure 15–6 The Web mail system displays a summary of the user mailbox on the Mailboxes.wssp page. Choosing the InBox link displays the contents of the mailbox.

Marilyn wants to start reading messages so she clicks the InBox link. Figure 15–7 shows the resulting page that lists the links for the waiting email messages.


test10@saintleo.edu Mailbox InBox - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Mailboxes Compose Rules Settings External Log Out
Display Filter: Search: Mark All
Read: Set Clear
Flagged: Set Clear
Delete
Copy to... Move to...
Mailbox Management
Redirect to... Forward to...
Done
Internet

Address http://hera.saintleo.edu:8100/Session/14658-lrEy6cZr4YGzgxSuAeq5/Mailbox.wssp?Mailbox=INBOX&

1 of 2 unread Mailbox "InBox" 2 selected

Display 30

Status	From	Subject	Size	Received
	MAILER-DAEMON@saintleo.edu	Undeliverable mail: Campus quantos surba..	1467	04-Jul-03
	MAILER-DAEMON@saintleo.edu	Undeliverable mail: Delorum sit ad infre..	1373	04-Jul-03

Figure 15–7 Links for the waiting email messages.

The test agent script that takes the next step for Marilyn needs to do some extra work to reflect the dynamic nature of this page. The email system displays a list of links that reflect the list of waiting email messages. The test agent script needs to parse the html to find the set of links and then choose a link to read the corresponding email message. First I will show you the snippet of test agent script and then I will break it down and provide a detailed explanation.

```
loc = "http://" + mailweb + "/Session/" + sessionid +\
"/Mailbox.wssp?Mailbox=INBOX&"

http.setUrl( loc )
http.setType( HTTPProtocol.GET )
```

```
   response = http.connect()
   if response.getResponseCode()<>200:
       raise Exception, "Email list response code: " +\
       str( response.getResponseCode() )

   f1 = "<A HREF=\"Message.wssp?Mailbox=INBOX&MSG="
   f2 = "\">"

   responselink = ResponseLinkConfig()
   responselink.setParameter( 'beginsearch', f1 )
   responselink.setParameter( 'endsearch', f2 )

   search = SimpleSearchLink()
   search.init( responselink )

   found = search.handle( response )

foundcount = \
found.getParameterValue("simplesearch.foundcount")

   if ( foundcount == 0 ):
       return

   foundlist = \
   found.getParameterValues("simplesearch.founditems")
   doc = foundlist.get( random.randrange(0,foundcount) )

   # Next trim the <a href= and > tags to find the message
number
   msgnum = doc[ len(f1) : ( len(doc) - len(f2) ) ]

   # Get the message
   loc = "http://" + mailweb + "/Session/" + sessionid + "/
Message.wssp?Mailbox=INBOX&MSG=" + msgnum

   http.setUrl( loc )
   http.setType( HTTPProtocol.GET )
   response = http.connect()

   if response.getResponseCode()<>200:
      raise Exception, "Get message response code: " + str(
response.getResponseCode() )
```

This snippet of test agent script loads the mailbox summary page and chooses a waiting email message at random. The following is a detailed explanation of this test agent script snippet:

```
loc = "http://" + mailweb + "/Session/" + sessionid +\
"/Mailbox.wssp?Mailbox=INBOX&"

http.setUrl( loc )
http.setType( HTTPProtocol.GET )
response = http.connect()
if response.getResponseCode()<>200:
    raise Exception, "Email list response code: " +\
    str( response.getResponseCode() )
```

This snippet of test agent script gets the contents of the Mailbox.wssp page for this user. The `response` variable points to an object containing the html contents of the Mailbox.wssp page. The page contains links to the email messages in the user's mailbox. Each message is a link to the Message.wssp page with parameters indicating the message number. For example, the following URL is for message number 11.

```
http://hera.saintleo.edu:8100/Session/15091-
hoPMIzFZBueeyLo2zwSy/Message.wssp?Mailbox=INBOX&MSG=111
```

We can use TestMaker's parsing function to find the message number.

```
f1 = "<A HREF=\"Message.wssp?Mailbox=INBOX&MSG="
f2 = "\">"
```

The f1 and f2 variables define the search parameters. f1 is the start of the search phrase and f2 is the termination.

```
responselink = ResponseLinkConfig()
responselink.setParameter( 'beginsearch', f1 )
responselink.setParameter( 'endsearch', f2 )
```

The `responselink` variable will point to a `ResponseLinkConfig` object that will parse the Mailbox.wssp page for URLs that begin with the contents of `f1` and end in the contents of `f2`. `ResponseLinkConfig` is an object provided in TestMaker's Tool object library.

```
search = SimpleSearchLink()
search.init( responselink )
```

The design allows you to search for more than one phrase at a time by using multiple ResponseLinkConfig objects. For this test, only one is needed. These commands connect the ResponseLinkConfig object to the response object created when we make the GET request to the host.

```
    found = search.handle( response )
```

Applying the search.handle method to the response object parses the html for the search strings. The found variable now holds a list of URLs that match the ResponseLinkConfig parameters. In other words, a list of message links from the Mailbox.wssp page.

```
 foundcount = \
 found.getParameterValue("simplesearch.foundcount")

    if ( foundcount == 0 ):
        return
```

The found variable points to a search object. Among other methods, it provides the simplesearch.foundcount methods to tell us how many message links were found. If there are no email messages in the mailbox, the test agent simply waits for a while and then repeats this function again.

```
    foundlist = \
    found.getParameterValues("simplesearch.founditems")
    doc = foundlist.get( random.randrange(0,foundcount) )
```

The test agent script randomly picks one of the message links from the list of found values. The doc variable now holds the URL to the chosen email message. The URL looks something like this:

```
http://hera.saintleo.edu:8100/Session/15091-
hoPMIzFZBueeyLo2zwSy/Message.wssp?Mailbox=INBOX&MSG=111
```

The test needs to dynamically build the URL to get the email message Web page since the session identifier and the message identifier contain dynamic values.

```
    msgnum = doc[ len(f1) : ( len(doc) - len(f2) ) ]
```

The script uses a little bit of Python magic to find the message number in the chosen link. This command uses Python's string manipulation function to trim the <a href= and > tags to find the message number.

The test agent script is now ready to get the email message by issuing an HTTP GET command to the web mail host.

```
loc = "http://" + mailweb + "/Session/" + sessionid+\
"/Message.wssp?Mailbox=INBOX&MSG=" + msgnum

http.setUrl( loc )
http.setType( HTTPProtocol.GET )
response = http.connect()

if response.getResponseCode()<>200:
    raise Exception, "Get message response code: " +\
    str( response.getResponseCode() )
```

The snippet of test agent script above requests the Web page containing the chosen email message. Figure 15–8 shows the browser view of the command.

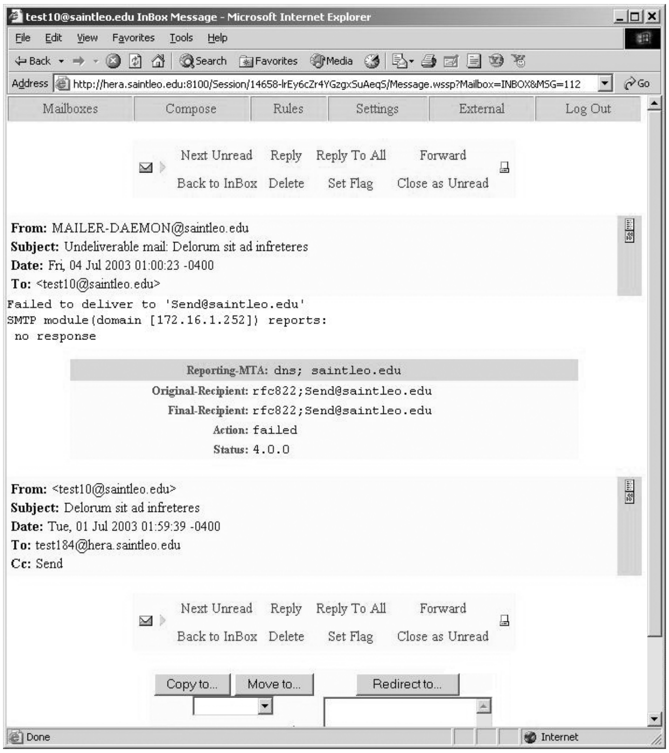
When new email is waiting, Marilyn immediately reads and deletes the email messages. The next step in the test agent script deletes the message we just retrieved.

```
loc = "http://" + mailweb + "/Session/" + sessionid +\
"/Mailbox.wssp?Mailbox=INBOX&MSG=" + msgnum +\
"&Delete=&"

http.setUrl( loc )
http.setType( HTTPProtocol.GET )
response = http.connect()
if response.getResponseCode()<>200:
    raise Exception, "Delete message response code:"+ \
    str( response.getResponseCode() )
```

The test agent script snippet issues an HTTP GET request to the Mailbox.wssp URL with a delete command in one of the parameters. The script dynamically builds the URL to include the session identifier and the message number. The Recorder automatically adds script commands to check the response code from the request.

The final step in building Marilyn is to implement her behavior to write email messages to other students. The Recorder watches us use a browser to


test10@saintleo.edu InBox Message - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://hera.saintleo.edu:8100/Session/14658-lrEy6cZr4YGzgxSuAeqS/Message.wssp?Mailbox=INBOX&MSG=112 Go
Mailboxes Compose Rules Settings External Log Out
Next Unread Reply Reply To All Forward
Back to InBox Delete Set Flag Close as Unread

From: MAILER-DAEMON@saintleo.edu
Subject: Undeliverable mail: Delorum sit ad infreteres
Date: Fri, 04 Jul 2003 01:00:23 -0400
To: <test10@saintleo.edu>

```
Failed to deliver to 'Send@saintleo.edu'
SMTP module(domain [172.16.1.252]) reports:
 no response
```

Reporting-MTA: dns; saintleo.edu
Original-Recipient: rfc822;Send@saintleo.edu
Final-Recipient: rfc822;Send@saintleo.edu
Action: failed
Status: 4.0.0

From: <test10@saintleo.edu>
Subject: Delorum sit ad infreteres
Date: Tue, 01 Jul 2003 01:59:39 -0400
To: test184@hera.saintleo.edu
Cc: Send

Next Unread Reply Reply To All Forward
Back to InBox Delete Set Flag Close as Unread
Copy to... Move to... Redirect to...
Done
Internet

Figure 15–8 Browser view of the HTTP Get Command.

click the Compose button. This action displays an HTML form for us to enter a new message.

```
loc = "http://" + mailweb + "/Session/" + sessionid +\
    "/Compose.wssp"

    http.setUrl( loc )
    http.setType( HTTPProtocol.GET )
    response = http.connect()
    if response.getResponseCode()<>200:
        raise Exception, "Send message response code:" +\
        str( response.getResponseCode() )
```

This command dynamically builds the URL by including the session identifier and gets the message composition page, Compose.wssp.

Next the test agent composes and sends a new message. Here is the code snippet to do so, followed by a detailed explanation.

```
http.setType( HTTPProtocol.POST )
body = ProtocolHandler.getBody( 'http' )
http.setBody( body )

body.addParameter('''FormCharset''', '''ISO-8859-1''')
body.addParameter('''Send''', '''Send''')
body.addParameter('''Cc''', '''Send''')
body.addParameter('''Bcc''', '''Send''')
body.addParameter('''DSN''', '''Send''')
body.addParameter('''MDN''', '''Send''')
body.addParameter('''OpenBook''', '''Send''')
body.addParameter('''desiredCharset''', \
'''ISO-8859-1''')
body.addParameter('''filled''', '''1''')
body.addParameter('''To''', '''test''' + \
str( random.randrange( 50, 300 ) ) + \
'''@hera.saintleo.edu''')

myLingo = Lingo()
body.addParameter('''Subject''', \
myLingo.getSubject( 4 ) )
body.addParameter('''Body''', myLingo.getMessage() )

response = http.connect( 0 )

if response.getResponseCode()<>200:
    raise Exception, "Post response code: " + \
    str( response.getResponseCode() )
```

This snippet of test agent code assembles and then issues an HTTP `POST` command to the Compose.wssp server function. The Recorder watches us receive the HTML form, type in the address of the recipient, the subject line and body of the message, and then click Send. The Recorder automatically writes the test agent script. We must modify the Recorders script to support the dynamic URLs needed to operate the email system for our test agent's account and to choose a recipient.

```
http.setType( HTTPProtocol.POST )
body = ProtocolHandler.getBody( 'http' )
http.setBody( body )
```

First, the script sets the existing `HTTPProtocol` object to issue a `POST` command to the host. Then it creates a new body that will contain the parameters of the message.

```
body.addParameter('''FormCharset''', '''ISO-8859-1''')
body.addParameter('''Send''', '''Send''')
body.addParameter('''Cc''', '''Send''')
body.addParameter('''Bcc''', '''Send''')
body.addParameter('''DSN''', '''Send''')
body.addParameter('''MDN''', '''Send''')
body.addParameter('''OpenBook''', '''Send''')
body.addParameter('''desiredCharset''', \
'''ISO-8859-1''')
body.addParameter('''filled''', '''1''')

body.addParameter('''To''', '''test''' + \
str( random.randrange( 50, 300 ) ) + \
'''@hera.saintleo.edu''')
```

I modify the Recorder's version of the `POST` command to dynamically address the email message to one of the test accounts. For the purpose of this test, Marilyn expects faculty members to use the email accounts named test1 through test49. So the test agent script randomly picks an email account from 50 to 300.

```
myLingo = Lingo()
body.addParameter('''Subject''', \
myLingo.getSubject( 4 ) )
body.addParameter('''Body''', myLingo.getMessage() )
```

One other change from the Recorder script is to dynamically create the message subject and body. For this, I use the Lingo object provided in TestMaker's Tool library. Lingo returns gibberish Latin-like words. Using Lingo reduces the risk that a cache manager will impact the test results because in this test every message contains unique content.

```
response = http.connect( 0 )

if response.getResponseCode()<>200:
    raise Exception, "Post response code: " + \
    str( response.getResponseCode() )
```

Finally, the `connect` method issues the `POST` to the web mail host and we are done. Marilyn then goes back and checks for more email messages.

This test agent script we just covered implements Marilyn's behavior. We used the Recorder to create a test agent script that uses the university Web Mail interface. The Marilyn test agent uses the browser interface to check for email and to send messages. She immediately reads and deletes all email messages in her email account. And every time she checks for messages, she sends at least one email message to one of her friends.

Result Analysis

The university email test environment uses a distributed set of test machines to generate load on email system and its supporting infrastructure. Each of the test agent scripts is run multiple times concurrently to generate increasing amounts of virtual users. Each test concludes by analyzing the logged results data to determine the throughput of the email system. Figure 15–9 shows how this process works.

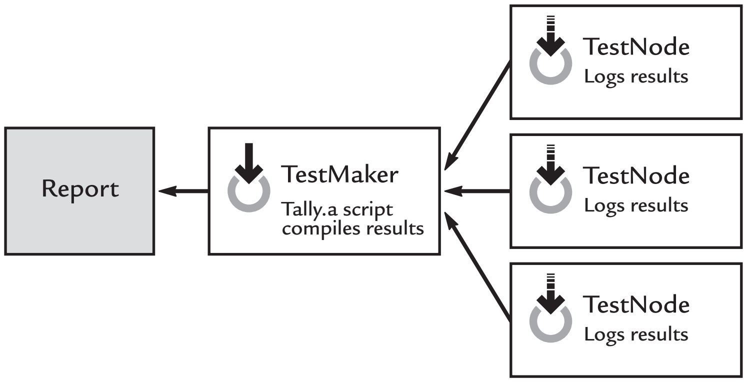


Figure 15–9 While the test runs the test agent scripts logs results data to a log file on the local TestNode machine. At the conclusion of the test, the TestNodes return their logged data to the central TestMaker console for analysis and reporting.

The test agent scripts create a simple comma-delimited data file with these fields:

- Time to complete a transaction
- Current system time

- Test agent archetype
- TestNode machine identifier
- Type of transaction

Once the result logs are returned to the TestMaker machine, the *tally.a* test script analyzes each of the log files and compiles a set of statistics.

- Average transactions-per-second
- Minimum duration of a transaction
- Maximum duration of a transaction
- Count of transactions completed by each user archetype
- Count of transactions that ended in an error condition
- Total time to run the test

The *tally.a* script runs automatically at the end of each test, so the test may be fully automated.

Summary

In this chapter, the needs of a major online university were met by implementing a user goal-oriented test of the capacity of the email system to meet their users' goals. This chapter discussed the needs of the university to conduct a multiple protocol test in an email environment, the choices of test tools and methodologies, describes in-depth the test agent code used to conduct the test, and showed how the test provided valuable information to the university so it may save time, energy, and resources while well serving its users.

Index

A

Administrators, 13
Agile programming, 11–13, 25
  and Micromax tool, 87
  test first precept, 166
Ant utility, 167
Apache
  Axis, 95
  Jmeter, 129
  license, 145
  SOAP, 141t, 235
  Tomcat, 95
  XML-RPC, 143
Availability, 389

B

Back end systems services, 7, 9f
  application, 7–8
  authentication authority, 7
  instant message server (IM), 8
  network operating system (NOS), 8
BEA WebLogic, 233, 422
  and "flapjacks architecture," 59
Beck, Kent, 167
BigDecimal data type, 236
Border Gateway Protocol (BGP), testing in flapjacks environment, 94
Borland JBuilder, 127
BPEL4WS, 10, 130
Browsers, 4
Business logic, 4
Business management styles, 99t–100t
  and effective testing design, 80, 99
    case example, 100–101

C

Cape Clear CapeStudio, 233
Click-stream measurement tests, 45
Client-side system-level tests, 16
Client/server architecture, 4, 124–126
  test automation, 126–127
CLR (Common Language Runtime), 293
Coincident, 111
Common Gateway Interface (CGI), 26
CommuniGate Pro, 449
Compuware OptimalJ, 127
  Object Management Group (OMG) Model Driven Architecture (MDA), 129
Concomitant, 111
Concurrent, 110, 385, 389
Connected software applications *see* Web-enabled applications

Cookies, 194
Cooper, Alan, 109

D

Desktop automation phase, 122–124
Desktop Software Development Lifecycle, 23–24
DHCP (Dynamic Host Control Protocol), 321
Disco (Web Services Discovery Tool), 305

E

eBay
  and flapjacks architecture, 59
  load pattern case example, 44
ECMA (European Computer Manufacturers Association), and CLR standards, 293
Empirix
  e-Load, 129
  e-Test, 127
Enterprise infrastructure, 5, 7f
  see also Back-end systems services; Web infrastructure services
ethereal, 255
Excite, 26
Extensible Markup Language (XML), 173, 217, 420
  integration opportunities, 217–218
  and .NET, 295
Extreme programming *see* Agile programming

F

"Flapjacks architecture," 57–58, 58f
  benefits of, 59, 93–94
  and security issues, 329
  strategies for implementation, 59–60, 60t
    targeting developers, 60–62, 62f
    targeting IT managers, 63–64
    targeting QA managers, 62–63
  and testing modules for functionality and scalability, 92–93
  see also Test agents
"Follow me home" software testing method (Intuit), 36
Functionality and scalability testing case example, 87–88, 88f
  and flapjack architecture, 92–93
    advantages, 93–94
  functional testing, 90–91, 90f
    and intelligent test agents, 91, 91f
  scalability testing, 91–92, 92f
  testing challenges, 90
  user view, 88
Functional testing, 90

G

Go-Mono, 317
Government regulation and software application requirements, 9
Grid computing, 104, 421
  see also Self-healing systems
"Grid" generation, 43–45
The Grinder, 129

H

Heath Insurance Portability and Accountability Act (HIPAA), 9
HMux (Caucho), 406
`hostdoc_decoder` function, 191, 200
HotMail, and flapjacks architecture, 59
HP OpenView, 384
HTTP test agents, 182, 182f
  `HTTPProtocol` object overview, 183, 183f, 192–193
  Sign-In Agent example, 182
    sign-in forms and cookie managed sessions, 194–202
  and TestMaker, 182–183
  Validator agent example, 182
    and JDOM commands, 213–214
    `ResponseLink` methods, 210–211

and scripting language commands, 211–212
server response data searched/parsed/validated, 203–214
TOOL commands for parsing, 212–213
Wanderer agent example, 182, 184
Python and Tool interaction, 184–193
HTTP/1.1 protocol, 405
HTTPS connection, 6
Hypertext Markup Language (HTML), 173
content-checking tests, 45
see also The Web
Hypertext Transfer Protocol (HTTP), 19, 173
basic authentication, 332–333
see also The Web

I

IBM
Rational Rose, 127, 129
WebSphere and "flapjacks architecture," 59
IL (Intermediate Language), 293
Inclusion Technologies, archetypal users for testing, 36–38
Infrastructure, 26
maintenance tips, 16
Integrated development environment (IDE), 9–10, 21
Internet Control Message Protocol (ICMP), 45
Internet Engineering Task Force, 177
Internet Relay Chat (IRC) service, 6, 8
Internet Software Development Lifecycle, 24–25
Interoperability programmers, 14
Interoperating software applications *see* Web-enabled applications
Intuit, "Follow me home" software testing method, 36
ISO-8859-1 encoding, 224

J

J2EE objects, 7
JAR (Java Archive Resource) file, and drivers, 283
Java
and Jython integration, 165
Mail API, 143t
omission of scripting model, 66
Page Flow, 130
Server Faces, 130
servlets (HTTP communication protocol handling), 7
see also Jython
Java Community Process (JCP), 166
Java Secure Sockets Extension (JSSE), 336
and TestMaker, 339–341
Java Server Pages (JSP), 173
Java Specification Request (JSR) 223, 166
Java Web Services Developer Package, 233
jCookie, 142t
JDBC (Java DataBase Connectivity), and TestMaker, 280–286
JDOM, 142t, 182, 213–214
Jini *see* Sun Microsystems
JNumeric, 142t
JOpenChart, 142t
JUnit, 166
case example, 168–169
and repeatable tests, 166–167
goals of unit test framework, 167
and `TestCase`, 167–168
and TestMaker, 169
case example, 169–172
JXTA *see* Sun Microsystems
Jython, 66, 67, 142t, 162–163
features/benefits
Bean property introspection, 165–166
built in data structures and list iteration, 163–164
dynamic variables and automatic typing, 164, 189
functions/methods/modules/classes first-class objects, 164–165, 188
Java expansion (scripting), 166
Java integration, 165, 166
quickness, 163

formatting, 74
import command, 188–189
and Java objects case example, 68–71
test agents creations case example, 154–159
and TestMaker, 144, 188

K

KeyNote Systems Internet weather report, 26

L

Lane, Geoff, 144
Latency, 389
Liberty Alliance, 10, 335
Lightweight Directory Access Protocol (LDAP), 8
in bookstore case example, 89, 89f

M

Mainframe Software Development Lifecycle, 22–23
Management styles *see* Business management styles
MaxQ, 142t
MDA (Model Driven Architecture), 129
Mercury Interactive LoadRunner, 127, 129
Mercury Interactive WinRunner, 124
Micromax Lifecycle, 83
categorizing problems, 83–85, 83t
prioritizing problems, 85–86, 85t
reporting problems, 86
evaluation criteria, 86–87
Microsoft
criticism of, 105, 219
Direct Internet Message Exchange (DIME), 421–422
DNA (Distributed iNternet Architecture), 293
IL (Intermediate Language), 293
.NET Framework, 220, 293–294
case example of mixed languages run by CLR, 294–295
CLR (Common Language Runtime), 293
current issues for design and testing, 315–316
and Disco (Web Services Discovery Tool), 305
document-style SOAP encoding case example, 298–300
and “flapjacks architecture,” 59
integration issues (case example), 295–297
interoperability issues, 295
Passport authentication, 331–332
and SOAP header values, 303–304
test agent case example, 307–315
and WSDL, 304–307
and WSML (Web Services Meta Language), 305
Network Monitor utility, 255
VB.NET, 316
Visual Studio, 124
and Web-enabled application protocols, 106, 302
Multiprotocol testing/email environment case example
project scope, 448–449, 449t–450t
result analysis, 476–477
test design, 451–452
resources, 451f
test environment installation and configuration, 455–456
Recorder activation (coding examples), 462–476
script creation for archetypes (coding examples), 456–462
Test Agent Files, 455t
test methodology, 452t–453t
university requirements, 447–448
user archetypes, 453–454

N

N-Tier architecture *see* "Flapjacks architecture"
NetMechanic monitoring service, 26
NetWatch, 255
Network Address Translation (NAT), 321
Network Associates, DOS and Windows fixes, 105

O

Object-oriented programming, 128
Object programmers, 14
Open source development and distribution, 25, 145–146
Orchestration programmers, 14

P

Peer-to-peer (P2P) technologies, 5
Performance, 389
Performance testing, 30, 33–35
- criteria definition, 44
  - click-stream measurement tests, 45
  - content-checking tests, 45
  - load pattern focus, 44
  - ping tests, 45
- key questions, 45
  - acceptable performance, 46
  - basic features, 46
  - failure rate, 46
- and test matrix template, 64–65
- tools, 52–53
  - tool kit, 429–431
- *see also* SPC (Scalability and Performance Criteria); Web-enabled application measurement tools; Web Services Performance Kit

Personal digital assistant (PDA), 19
Ping tests, 45
Platform dependency reductions, 10
Presentation code, 4
"Price/performance" ratio, 110
Procedural programmers, 13–14
Programming, 13
Programming techniques
- agile, 11–12
- "inspired wave," 12
- problem solving styles, 13–15, 13f, 15f

Public key infrastructure (PKI), testing in flapjacks environment, 94
PushToTest, 129, 145, 172
- online support services, 46
- and test scripts, 74–75
- TestNetwork test node appliance, 392
- use of WAPS, 55–56
- *see also* Web Services Performance Kit

Python programming language, 67, 162
- case example (TOOL and Python interaction), 184–193
  - loops, 192
  - spaces to denote functions/groups of commands, 190

Q

Quality of Service (QoS) testing, 30

R

RadView
- Web FT, 127
- WebLoad, 129

Resources
- security issues, 341
- SOAP and WSDL standards, 256

S

Scalability, 9
- impact of tools, 428–429
- test results analysis
  - actionable knowledge, 380t
  - log file, 379
  - pitfalls of quick overview, 377, 380

test goals and actionable knowledge case examples, 381–383
test results analysis (“Big Five” problem patterns), 383, 388–389
component problems, 386–387
concurrency problems, 384–386
contention problems, 387–388
crash recovery problems, 388
resource problems, 383–384
test results analysis metrics, 389–391
testing, 30, 33–35, 92
test goals, 344–345t
testing (stock trading case example)
good performance definition, 345
impact of test agents on scalability, 374–376
logging component implementation (master component), 372–373
master component implementation, 365–366
master component implementation (cleanup), 371
master component implementation (run), 368–370
master component implementation (setup), 366–368
mix of user archetypal behaviors, 356f
placement of intelligent test agents in infrastructure, 347
property files, 371–372
requirements identification, 350, 351t
results overview, 376–377
system infrastructure, 346, 346f
tally component, 394–399
test agent environment flow, 352–355, 353f
user archetype code modules, 356–365
user archetypes identification, 348–350
see also SPC (Scalability and Performance Criteria)
Script writers, 13–14
Scripting language, 66
common run-time environment usage, 67–68
downside, 66
examples, 67t
formatting test scripts, 74
and Java/Visual Basic, 66
and test agent logic, 71
and test agent script case example, 71–74
see also Jython
SCRUM *see* Agile Programming
Security Assertion Markup Language (SAML) protocol, 8, 10, 335
artifacts, 259
in bookstore case example, 89f, 90
Security issues
functional testing and security infrastructure, 320
impact of September 11th, 319
need for federations, 9
network segments and subnets, 323–324
load balancers, 324, 324f
security by routing, 320–322, 321f
SOAP problems, 333–334
and new technologies solutions, 335–336
TestMaker case example (U.S. Navy), 337–338
transport security, 325
Virtual Private Networks (VPNs), 322–323
performance and scalability limitations, 323
see also Hypertext Transfer Protocol/basic authentication; Microsoft/.NET Framework/Passport; SSL protocol
Segue SilkTest, 127, 129
and Dileep’s Dilemma, 393
Self-healing systems, 106–108, 107f
Service level agreements (SLAs), 101
terms, 101t–102t
additional requirements, 102–103
guarantees, 102
see also WSLA (Web-enabled application Service Level Agreement)
Simple Mail Transfer Protocol (SMTP), 19
Simple Object Access Protocol (SOAP), 19, 218, 221, 221f
call types, 245
document-style SOAP messages, 248–253
formatted XML data, 254

and .NET, 297–300
RPC (remote procedure call) style SOAP calls, 245–248
calls construction, 242
custom data types via TestMaker, 244–245
parameter encoding via TestMaker, 244
types of parameter encoding, 242–243, 243t–244t
creation stages, 300
scalability and performance problem points, 300–302, 301f
document-style encoding, 421, 422–423
encoding styles, 421–424, 423f
impacts, 419, 428–429
header elements, 302–303
interoperability issues, 232–235, 297
multistep process protocol, 228, 228f
and .NET case example, 296–297
popularity, 419–421
Remote Procedure Call Literal encoding, 421
Remote Procedure Call (RPC), 421, 422
Section 5 encoding, 421
security issues, 333–334
security solutions, 335–336
and SSL, 330–331
testing case example (Elsevier), 424
"moving target" issues, 445–446
and TestMaker, 424–427
and Userland, 227–228
validating response data, 255
writing agents (via TestMaker), 255
and WSDL, 228–229
Simultaneous, 111
Smalltalk, 167
SMTP email service, 6
SOAP *see* Simple Object Access Protocol
SOAPBuilders, 232–233
Software development
architecture, 4
historical generations of, 219, 220f
axioms, 3
Client/Server automation phase, 124–127
Desktop automation phase, 122–124
"firsts," 9–10
historical phases of development/digestion, 4, 19–20
error-handling code focus, 20–21
objectives, 1
productivity tools, 129–130, 130t
waves of automation, 121, 122t
see also Programming techniques; Web-enabled applications
Software development community, sharing information, 2–3
Software development obstacles, 1–2, 10–11, 18–19
development vs. QA vs. IT management issues, 18
homogeneity quest, 15–17
invalid test strategies, 17–18, 20
management directives vs. developers' strengths, 12–15
small implementations vs. major overhaul (new release "solidity" expectations), 11–12
see also Software testing/historical paradigms
Software testing, 53
boundary timing testing, 64
criteria, 41–42, 42f
data collection issues, 96
goal-driven user archetypal testing, 18
historical paradigms, 21
Brute Force Testing Method, 22
see also Desktop Software Development Lifecycle; Functionality and scalability testing case example; Internet Software Development Lifecycle; Mainframe Software Development Lifecycle
invalid data, 179
boundary data errors, 180
too few/many parameters, 179
wrongly formatted data, 180–181
wrongly ordered data, 179–180
invalid test strategies, 17–18, 20
lifecycle (stateful systems), 261–263, 262f
and misleading result scenarios, 391
Dileep's Dilemma, 393
diminishing returns, 394
hidden error, 392–393
node problem, 391–392

modern methods, 25–26, 76
click-stream testing, 26–27
functional system testing, 28–29, 29f, 30
quality of service (QOS) testing, 30
scalability and performance testing, 30
unit testing, 27–28
see also SPC (Scalability and Performance Criteria)
privilege testing, 64
rapid application testing, 60–61
regression testing, 64
for single user, 35–38
speed testing, 64
stateful testing, 64, 257
and system design issues, 79–80
transparent failure issues, 97
invalid inputs/responses, 98
limited database connections, 98
limited network bandwidth, 97–98, 98t
load balancer as single point of failure, 98–99
and The Troika, 18, 131–133
universality of, 77
and user behavior modeling, 78, 78t–79t
see also Business management styles; Client-side system-level tests; Inclusion Technologies; Intuit; Test agents; Test automation lifecycle; Test automation tools; Web-enabled application points system (WAPS)
SPC (Scalability and Performance Criteria), 108–109
case example, 111–113, 112f
modeling techniques, 116–117
SPI calculation, 113–114, 113f
SPI usage, 114–115
test description, 119–120
user archetypes, 115–116
discourage failure, 110
jargon specificity, 110–111
reward increased price/performance, 110
reward success, 109
reward time savings, 109–110
and SPI (scalability and performance index), 111
SSL (Secure Sockets Layer) protocol, 325
common problems, 329
eCommerce scenario case example using TestMaker, 327–329
features, 325–326
in SOAP environment, 330–331
X.509 certificates, 326
Stateful systems, 257
benefits, 257–258
challenges, 258
cookies, 258
SAML Artifacts, 259
URL rewriting, 258–259
establishing state (techniques for)/discussion system case example, 263–264,
263f, 278–280
command protocol to establish initial state, 265–266, 265f
command protocol/read from file technique, 266, 267f
state issues for scalability and performance test, 264
testing environments, 264–265
TestMaker Prepare_Setup_Agent for *commands* file (code example), 267–273
TestMaker Setup_Agent executing commands (code example), 273–278
testing, 64, 257
horizontal and vertical scalability, 259–261, 260f, 261f
testing lifecycle, 261–263, 262f
Sun Microsystems
and Jini, 104
and JXTA, 104
and Micromax Lifecycle, 83
ONE Studio, 124
Sun Community Server (SCS) development project, 140
Symantec
and case example of management style/testing issues, 100–101
and Micromax Lifecycle, 83
Synchronous, 111

System Network Monitor Protocol (SNMP)
standard, 384

T

Tation, 109
tcpdump, 255
Test agents, 30–31, 31t, 32–33
  automated test agents, 39
  checklist, 31–32
  data collection considerations, 96
    data based on test criteria, 96–97
    data storage issues, 97
    transparent failure issues, 97–99
  in “flapjacks” environment, 64
  and generating meaningful data, 75
  intelligent, 181–182
    archetype creation, 38–39, 42–43, 75
    and grid computing, 104–105
    and session tests, 181
    use in functional testing, 91, 91f
    use in scalability testing, 92, 92f, 343
    see also HTTP test agents
  maintenance issues, 65–66
  reporting method, 31, 32
  and scalability and performance testing, 33–35
  “stateful” (case example), 117
    action, 117
    analysis, 117
    conditional recurrence, 117
    establishment, 117
    setup, 117
  states identification *see* UML
  test process, 31, 32
  traits, 50, 51t
  see also Scripting language
Test automation lifecycle, 133, 134f
  goals/systems components/user archetypes identification step, 134–135
  ITE (integrated test environment) to construct tests, 135
  record test script to drive browser interface, 135
  remote monitoring over time, 136–137
  run test/native protocols, 136
  scalability test/multiple test machines, 136
  show tests graphically, 136
  statistical report of results, 136
  test scripts into test framework for test suite, 135
  university email case study, 133
  write test script to drive SMTP and POP3 interfaces, 135
Test automation tools, 128–129
  personal quest for, 140–141
    toolbox, 141t–143t
TestMaker, 3, 57, 71, 129, 139, 172
  access to, 144–145
  Agent Recorder, 159, 462
    browser configuration requirements, 159–160, 160f
    case examples, 160–162, 161f, 462–476
  architectural view, 183f
  and database connectivity, 280–286
  installation (Windows or Linux), 147
    files/directories, 147t
  and Java keytool JSSE, 339–341
  Lingo utility, 286–287
    case example, 287–289
    `getMessage` function, 290
    `getSubject` function, 290
    `Lingo_meister`, 289, 290
  in .NET environment (case example), 307–315
  property files, 371–372
  revisions to, 143–144
  running (launcher scripts), 148
    graphic environment, 148–150, 149f
  running (test agents), 149f, 150–153, 150f, 151f, 152f
  test agent building steps, 146
  test agents creation (New Agent Wizard), 153–159, 153f, 154f, 183, 188, 255
    Jython code sample, 154–159
  Test Object Oriented Library (TOOL), 182
  Tool protocol handler objects (case example), 170–172
  see also JDOM; JUnit; Jython; Multiprotocol testing/email environment case example; Stateful systems;

2Wire; WSDL (Web Services Description Language)
“Thick”/“thin” application architecture, 173–174
“Third-party opportunities,” 105
The Troika *see* Software testing
2Wire, 401–402
CMS (component management system), 402, 404f
benefits, 403
concurrency testing, 407–408
concurrency testing via TestMaker, 408–409, 409f
roles in TestMaker environment, 409
system architecture, 404–407, 405f, 407f
CMS test project/performance and scalability audit via TestMaker, 410, 411
operations and planned frequencies/test agent mixture, 412t, 413, 413t
real-world device emulation, 411
setup steps, 413–414
test environment considerations, 414–415, 414f
test environment constraints, 416–417
test environment devices/tools/settings, 415t
transaction categories, 412t
user-goal orientation, 411
vertical and horizontal scalability measures, 411

U

UDDI (Universal Description, Discovery and Integration), 220–221
and interoperability issues, 232
UML (Unified Markup Language), 118, 129
and actions/states identification, 118
actors, 349
use case example, 118, 118f
User archetypes, 348–350
Userland Software, 227
Users/super users, 13

V

Visual Basic, omission of scripting model, 66

W

The Web, 173
freewheeling nature of, 177–179
HTTP/HTML environment (bank funds transfer service example), 174
browser noncompliance with standards errors, 177
caching problems, 178–179
common errors, 177
customer view, 174–175, 175f
HTTP `GET` and `POST` commands, 175–177, 175f, 176t
invalid data errors, 179–181
session problems, 181
see also HTTP test agents
and stateless protocols, 257
technologies for putting something on, 173
and Web Services, 217
and XML, 217
Web-enabled application measurement tools, 46–47
maintenance issues, 64–65
measurement dimensions
availability, 49
concurrency, 49
latency, 49
performance, 49–50
see also Web-enabled applications points system (WAPS); Web rubric
Web-enabled application points system (WAPS), 53–54
automation of, 54–55
and developers, 56–57
measurements, 54t
scalability, 56
Web-enabled applications, 5, 5f, 127
client application expectations, 127
environment, 19
flexibility, 21

framework, 57
“flapjacks architecture,” 57–59, 58f
“grid” generation, 4–5
lifecycle issues, 79–80
development perspectives, 81
impact of special interests, 81–82, 82f
see also Micromax Lifecycle
platform independence, 127
productivity gains, 19–20
scalability, 21
statelessness, 127
test automation, 128–129
test data collection points, 94, 95f, 96
business rules, 95
database, 95
serializers, 95
SOAP parser, 95
Web server, 94
XML parser, 95
types, 51–52
see also Integrated Development Environment (IDE); Security issues; Stateful systems
Web infrastructure services, 5, 7f
click over from partner site, 6
need more stock, 6–7
partner confirms order, 6
send pager message, 6
sign-in function, 5–6
team discussions (IRC service), 6
Web rubric, 47, 50
advantages, 48
application case example, 47t
features, 49
Web Services model, 218–221
application-to-application communication via common language, 219
interoperability problems, 231–232, 237
date data types, 236–237
definition, 232–234
discovery, 232
floating-point and decimal data types, 235–236
request/response, 234–235
scalability techniques, 229–231
and WSDL (Web Service Description Language), 218, 228–229
Web Services Performance Kit, 430–431, 445
installation, 431–432
PerfCheck, 435–436
master.a script, 436–442
properties.a script, 436
test threads, 442–445
questions, 429–430
start-up, 432
Test Web Service (TWS), 432–435
WebTrends log analysis utility, 26
Wily Introscope, 129
WinPerl Guido, 124
World Wide Web Consortium, 177
WS-I (Web-enabled application Interoperability), 10, 335–336
WSDL (Web Services Description Language), 218, 237
case example using TestMaker, 237–241
test agent and WSDL definition, 241–242
and interoperability issues, 233
WSLA (Web-enabled application Service Level Agreement), 103
WSML (Web Services Meta Language), 305

X

XML *see* Extensible Markup Language
XML Key Management Specification (XKMS), 335
XML-RPC protocols, 6–7, 218, 221–222
case example, 222–224
and TestMaker test agent script, 224–226
limitations, 227
XML Schema, 335
XML Signature, 335
XP programming *see* Agile programming

Y

Yahoo, and flapjacks architecture, 59