UNIVERSITY OF CALGARY

In-Container Testing for Web Portal Applications

by

Wenliang Xiong

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

September, 2006

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled " In-Container Testing for Web Portal Applications " submitted by Wenliang Xiong in partial fulfilment of the requirements of the degree of Master of Science.

Supervisor, Dr. Frank Oliver Maurer, Department of Computer Science

---

Craig Taube-Schock, Department of Computer Science

Dr. Victoria Mitchell, Haskayne School of Business

Date

Abstract

In order to test interactions between container and components, a mechanism is required to intercept and control all information passed through interfaces before it reaches component code. Such an approach provides developers with a chance to set up testing scenarios by manipulating data sent to component code, and validates component-generated data before it is returned back to the container. This approach is named In-container Testing (ICT).

A proof-of-concept tool, namely WIT, was developed to support testing for interactions between JSR168-compatible Portlets and their container built by IBM. WIT allows developers to write automated in-container test cases for web portal applications. Using Aspect technology, the test code is injected into the application code allowing the tests to run in the same environment as the portal application. In this paper, we present the overall testing approach, design & implementation of WIT as well as a usage scenario.

.

Publications

Xiong, W., Bajwa, H., and Maurer, F. (2005) **WIT: A Framework for In-container Testing of Web-Portal Applications,** *Proceedings of International Conference of Web Engineering (ICWE 2005)* LNCS, Volume 3579, Jul 2005, Pages 87 – 97

Bajwa, H., Xiong, W., and Maurer, F. (2005) **Evaluating Current Testing Processes of Web-Portal Applications**, *Proceedings of International Conference of Web Engineering (ICWE 2005)* LNCS, Volume 3579, Jul 2005, Pages 603 - 605

Acknowledgements

I would like to thank Dr. Frank Maurer for providing the initial idea for this thesis, for his kindly support and guidance.

I would also like to thank Harpreet Bajwa, Carmen Zannier, Ruth Ablett, Lawrence Liu, Thomas Chau, Chris Mann, and all the other EBE group members for their support and valuable comments and suggestions.

Dedication

This work is dedicated to my lovely daughter, Anik Xiong, and my wife.

Table of Contents

List of Tables

List of Figures

List of Acronyms

J2EE	Java 2 Platform Enterprise Edition
API	Application Program Interface
EJB	Enterprise Java Bean
COM	Component Object Model
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JVM	Java Virtual Machine
WWW	World Wide Web
WAS	WebSphere Application Server
WPS	WebSphere Portal Server
JSP	JavaServer Pages
ICT	In-Container Testing
SSO	Single Sign-On
WIT	Web Portal In-container Testing Framework

CHAPTER ONE: INTRODUCTION

With traditional software development approaches, a large percentage of software systems have been delivered late and over budget. They usually have low quality levels and high maintenance costs [Bosch 2000]. Component-based software development is a promising approach for solving these problems. It has been considered one of the three great revolutions in computing technology during the past fifty years [Maurer 2000]. In this approach, a software product is assembled from pre-fabricated components rather than from scratch. The reuse of existing components can reduce the development time and cost. Because it offers significant improvements in productivity, component-based development is becoming the mainstream of enterprise application development in the commercial industry. An entire industry is growing up to support software component technology [SEI2000]. At present, Sun Microsystem's Enterprise JavaBeans (EJB) [EJB2006] and Microsoft's COM [COM2006] are two of the most well known component-based products in the industry.

A component, according to Szyperski's definition, is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [Szyperski1998]. This definition describes a component as a self-contained set of code. Each component provides a set of interfaces through which a user can invoke its operations. If a component relies on its outside environment to provide proper functions, such dependencies must be declared explicitly. Furthermore, this definition emphasizes

the component's deployment independency as well as its ability to be integrated with other components.

A component framework, or component infrastructure, is a collection of rules and interfaces (contracts) that govern the interaction of components plugged into the framework [Szyperski 1998]. It is a special type of component that offers infrastructure services used by other components of the system. This includes such services as an event-handler, a database management, authorization management, and so on [Kim 2005]. Software vendors build and sell these commercial products. For example, IBM's product, WebSphere Application Server [WAS 2006], consists of several component frameworks to support other components, including EJBs, Servlets. This product is built in compliance with the J2EE specification. J2EE [J2EE 2006] is the industry standard for developing portable, robust, scalable and secure server-side Java applications. In this specification, component frameworks are also called containers such as EJB container, etc. Containers host a number of component instances, and provide system-level services such as persistence, security, distribution, transaction management etc, for their execution.

A typical scenario involving a client acquiring component services is described as follows: 1. The client sends a request for the component services, 2. The container first handles the request, and then forwards the request to the component, 3. In collaboration with container, the component handles the request and generates a response for the container, 4. The container forwards the component's response to the client. In this scenario, the container interacts with the component by: 1. Forwarding information from

the client side to the component, 2. Sending the information generated by the component back to the client, 3. Providing container services to the execution of the component. These interactions happen through the interfaces defined by the container.

Containers save substantial development effort on the part of application developers. They are freed from having to implement the services common to all components, and thus can concentrate on implementing the more important business logic. In the following sections of this thesis, I use the term ‘container’ for component framework, because my research project is related to a product built on the J2EE-compliant platform, WebSphere Portal Server [WPS2005].

While the advantages of component-based technology are obvious [Bass 2000], problems associated with the use of this technology have been reported in [Szyperski 1998] and [Clements 2001]. One of these problems is how to analyze a component’s behaviour from its client’s point of view. Since a component lives in container, when viewed from outside by a client who requires the component’s services, the functionality of a component is augmented by a container provided system-level service. In other words, the services a client receives are the combination of the services that a component provides and the services that a container provides. So when reasoning about a component’s behaviours, it is important to consider each component in conjunction with its hosting container. [Sridhar 2006]

A component may behave correctly as expected, or otherwise incorrectly. An incorrect behaviour causes a failure of a component's functionality. When a failure is observed on the client side, it is difficult to identify the precise origin of the failure because the failure may arise from the component being incorrectly implemented, the defective container or incorrect interactions between containers and components.

In industry, a typical scenario for building component-based applications consists of two steps: 1. Procuring commercial containers, and 2. Building new components in compliance with contracts defined by containers. In this scenario, professional software vendors should have thoroughly tested their products, the containers. The chances to cause failures of components because of malfunctioned containers are low. As discussed above, there are two more reasons that may be responsible for such failures. One is the incorrect implementation of components. Another is the incorrect interaction between containers and components. At present, the former issue can be solved with existing tools that are discussed in the chapter on related works. This thesis focuses on addressing the latter issue. It may not be possible to predict entirely how a component interacts with its container because the container and its configuration are unknown until deployment. Unpredictable changes in the container environment may occur when the component is deployed into a new container. Such changes may cause a component to fail. For example, an EJB container is responsible for establishing database connections for its components, EJBs. A container with incorrect database connections apparently will cause the failure of EJBs because EJBs cannot obtain the correct data. This means the successful

functionalities of EJBs achieved in one container may not be reproduced in other containers.

Software testing is the process used to discover the existence of faults in the software. The testing process starts from preparing an initial state and executing the code under testing. The process ends with checking the state generated by the execution of the code under testing. If the final state is the one expected, the code passes the test; otherwise the code fails. Testing is a critical activity to ensure the quality of software products. As discussed above, in some cases, failures of components surface only in the target container environment. To find out such failures requires an approach to test the composition of container and component. Existing testing tools and approaches are insufficient when performing such test. For example, [JUnit 2005] is a unit-testing tool that tests functionalities of a component piece by piece. On the one hand, an effective unit test ensures a component of code logic is correct, and thus reduces the rate of defects. On the other hand, the success of a unit test still cannot guarantee that a component's functionalities will work properly in a specific container, since the testing of a component is separated from the container in such approaches. More discussions about the limitation of existing testing tools are given in the Related Work section.

In most cases, the source code of a commercial container is not accessible. It is a black box from a developer's point of view. Testing such a container is out of the scope of this thesis. Testing the composition of containers and components can happen either inside or outside of a container. For the latter, the testing process starts from sending a simulated

service request, such as an HTTP request, to the target component and then comparing the component's response to the expected result. Any variation indicates the existence of failures. Even though this approach is able to identify that failures occur, some issues exist. For instance, the information it provides is not accurate enough to narrow down the scope of failures. A discussion of issues with this approach is given in the following sections. The former approach now seems more promising. In this approach, test code resides and executes inside a container. In this thesis, we refer to this approach henceforth as in-container testing (ICT).

The motivation for my research work arises from the above argument. That is, it centers around the question of how to perform a feasible test for component-oriented applications. This question originally comes from one of our industry partners, who is building component based web Portal applications [Wege 2002] using IBM's WebSphere Portal Server [WPS 2005]. The company reported

1) Unexpected deployment related errors and
2) A lack of an automated way to test access to sensitive portal resources.

In addition to solve the problems specifically related to the company's environment, this thesis proposes a novel approach, in-container testing (ICT), to perform functional unit tests [Section 2.3.2.3] against components when they provide service in a real container. As a proof-of-concept tool, the Web Portal In-container Testing Framework (WIT) is built to demonstrate the feasibility of this approach.

In brief, the objectives of this research are described as follows:

1. To develop an approach to test components deployed and executed in a container.

2. To implement a tool as a proof-of-concept.

3. To test the interactions between a container and components

4. To perform a qualitative comparison with other related approaches and tools. A qualitative comparison with other approaches and tools helps illustrate the uniqueness of WIT.

The rest of the thesis is organized as follows.

Chapter 2 is a description of the background of the WIT development, including an overview of Web Portal technology and software testing technology. It also describes the infrastructure of IBM's Web Portal product, WPS; Chapter 3 consists of the related work. The requirements for WIT are explained in Chapter 4 in detail. Chapter 4 also describes how this tool has been designed. Following the WIT design are WIT usage scenarios in Chapter 5 that explains how WIT can be applied in a real project. A qualitative comparison to other related approaches and tools is presented in Chapter 6. Finally, Chapter 7 discusses the future work and concludes this thesis.

CHAPTER TWO: BACKGROUND

The purpose of this chapter is to provide the background knowledge that is required to understand WIT, an in-container testing tool for applications built on IBM Websphere Portal Server. This tool applies two kinds of software technologies, Web Portal and Software Testing. To explain the former, this chapter begins with a general outline of Web Portal technology. This is followed by an introduction of the Web Portal technology standard in Java, JSR 168, and a specific introduction of the IBM WebSphere Portal Server, a product compliant with this standard. Next, this chapter presents a general introduction of software testing.

2.1 Web Portal Technology

A [Web 2006] Portal, or Portal in brief, is a popular Web technology in industry. A large number of software vendors provide this technology, including IBM and Microsoft. From a technical perspective, a Portal is a single integrated point of comprehensive, ubiquitous, and useful access to information (data), applications, and people [Saha 1999]. In other words, a Portal provides a single point of entry, a single point of access, and a single point of information exchange [Tushar 2002].

Figure 2.1 displays a typical Web Portal page from [MyYahoo 2006], a customizable web page with news, stock quotes, weather, and many other features. This Portal page is divided into a number of small windows. Each window displays the content of a specific topic, such as weather, news, etc. The content of each window is editable by clicking the

'Edit' button at the top-right corner. For example Figure 2.2 shows the page for editing weather information. On this page, a user is able to add weather forecast information for other cities, or change the metric unit of temperature. Figure 2.3 shows a page to allow a user to organize his/her Portal page. For example, the user is able to change the layout from two columns to three.

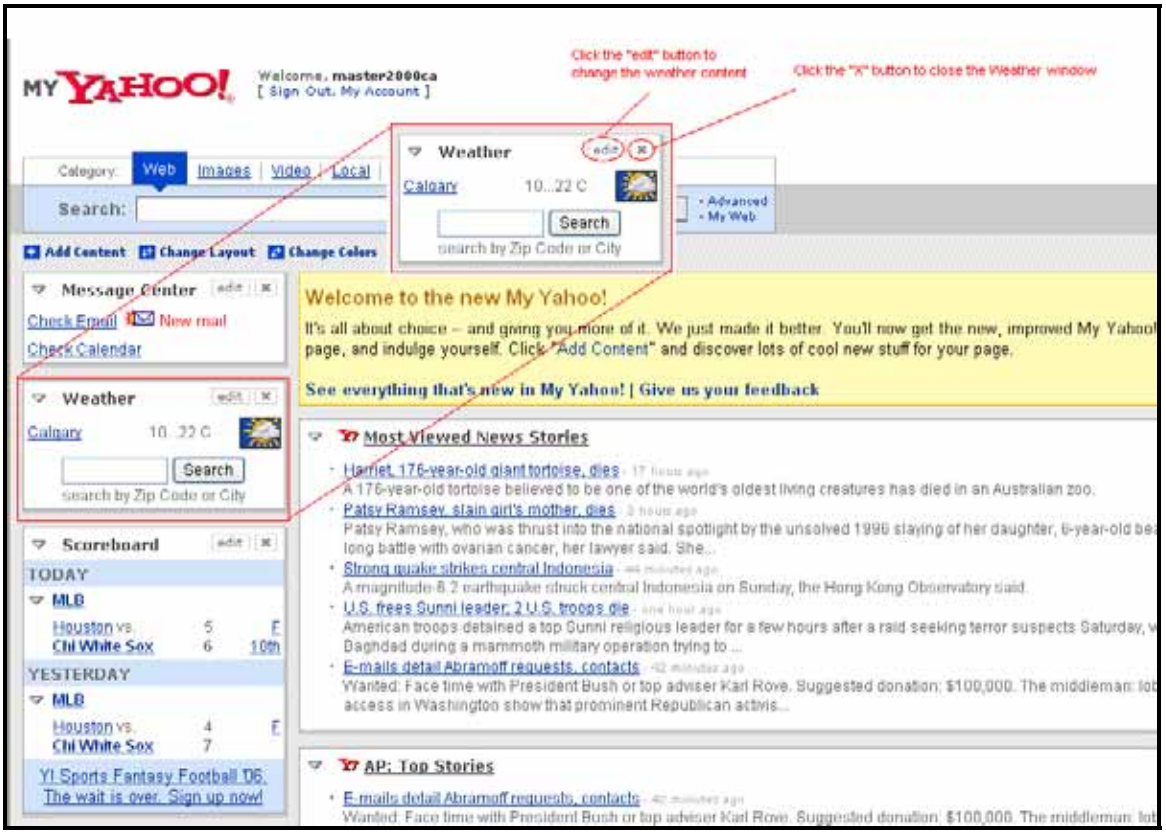

Figure 2.1 a typical personalized Portal page from My Yahoo

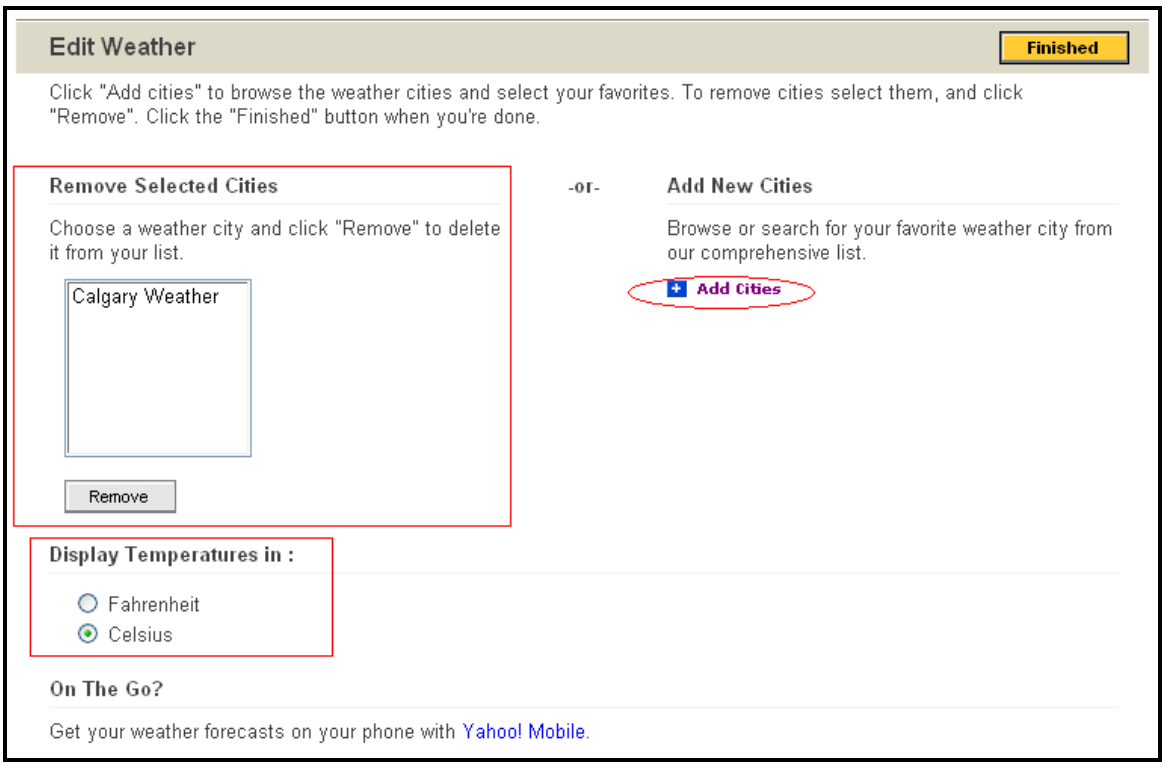

Figure 2.2 Edit the Weather Page

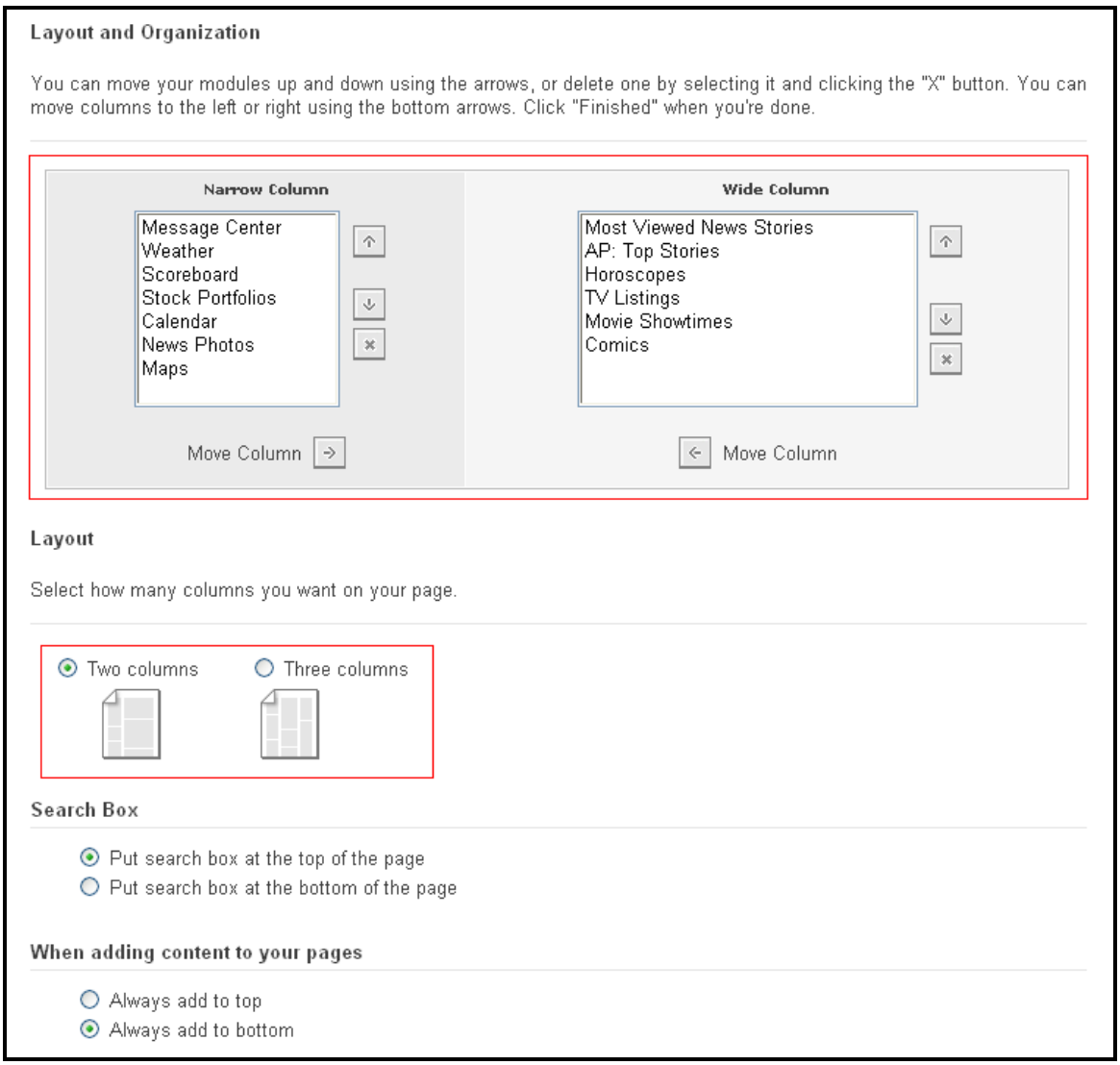

Figure 2.3 Change the layout of Portal pages

A typical Web Portal offers the following key features:

- **Personalization:** As demonstrated above, a Web Portal is able to identify different users and provides them the ability to customize the presentation of content.
- **Content Aggregation:** Figure 2.1 shows a Web Portal page, which aggregates different content from different sources, such as news, weather, etc. An enterprise

internal Portal usually provides employees aggregated organization-specific information from different applications.

- **Single Sign On (SSO):** Web Portal aggregates contents from a range of back-end systems, which have different authentication and authorization mechanisms. SSO enables a user access all these systems with a single action of user authentication and authorization. SSO reduces human error because a user does not need to manage multiple credentials for different systems.

2.2 Web Portal Technology in Java

A large number of software vendors are involved in building Web Portal products. Even though a Web Portal can be built using any Web based technologies, as depicted in table 2.1 below, popular Web Portal products in industry generally support either the J2EE, .NET platform, or both. This thesis solves issues involved in IBM's WebSphere Portal Server, a J2EE-based product. Hence, this thesis focuses on the Web Portal technology in Java area.

Vendor	Product	Supports
IBM	WebSphere Portal Server	J2EE
iPlanet	iPlanet Portal Server	J2EE
SAP	mySAP Enterprise Portal	J2EE and .NET
Oracle	Oracle 9iAS	J2EE
Computer Associates	CleverPath Portal	J2EE and .NET
PeopleSoft	PeopleSoft Enterprise Portal	J2EE

BEA	WebLogic Portal and Personalization Server	J2EE
JA-SIG	uPortal	J2EE
Microsoft	SharePoint Portal Server	.NET

Table 2.1: A list of popular Web Portal vendors and products.

2.2.1 JSR 168

Prior to the release of Java Portal technology standards, almost all Portal products had their own proprietary component to generate small window content that in turn is integrated into a whole Portal page. For example, iPlanet used a component called Provider to create pluggable Portal components, IBM had IBM Portlets, SAP had iViews. To solve the problem of component compatibility between different portal products, a Java specification, [JSR 168] was introduced to define a common API and infrastructure in order to provide facilities for personalization, presentation and security.

2.2.1.1 Concepts.

Below are some basic concepts introduced in this specification.

- **Portlet.** In JSR 168, a component is defined as a [Portlet]. A Portlet is a Java technology based web component, managed by a Portlet container, which processes requests and generates dynamic content. Portlets are used by Portals as pluggable user interface components that provide a presentation layer to information systems. [JSR 168]
- **Fragment.** The content generated by a portlet is also called a fragment. A fragment is a piece of markup (e.g. HTML, XHTML, WML) adhering to certain rules and can

be aggregated with other fragments to form a complete document. The content of a portlet is normally aggregated with the content of other portlets to form the portal page. [JSR 168]

- **Portlet Container.** A Portlet container runs Portlets and provides them with the required runtime environment. A Portlet container contains Portlets and manages their lifecycle. [JSR168 2005].
- **Portal/Portal Server.** A portal server (or portal for short) is responsible for integrating the fragments generated by the portlets into a complete page.

  A portal and a portlet container can be built together as a single component of an application suite or as two separate components of a portal application [JSR168 2005]. In this thesis, we treat the container as separate from a portal.

2.2.1.2 Portal page request sequence

Figure 2.4 illustrates a typical client request for a Portal page that interacts with multiple interfaces defined by a Portal server. The Portal server completes the client request for the Portal page by first retrieving the Portlets written by the developer for the current page. After this, the Portal server invokes the Portlet container for each Portlet. The Portlet container calls the Portlets via the Portlet API. The final Portal page presented to the client represents an aggregated content generated by several Portlets.

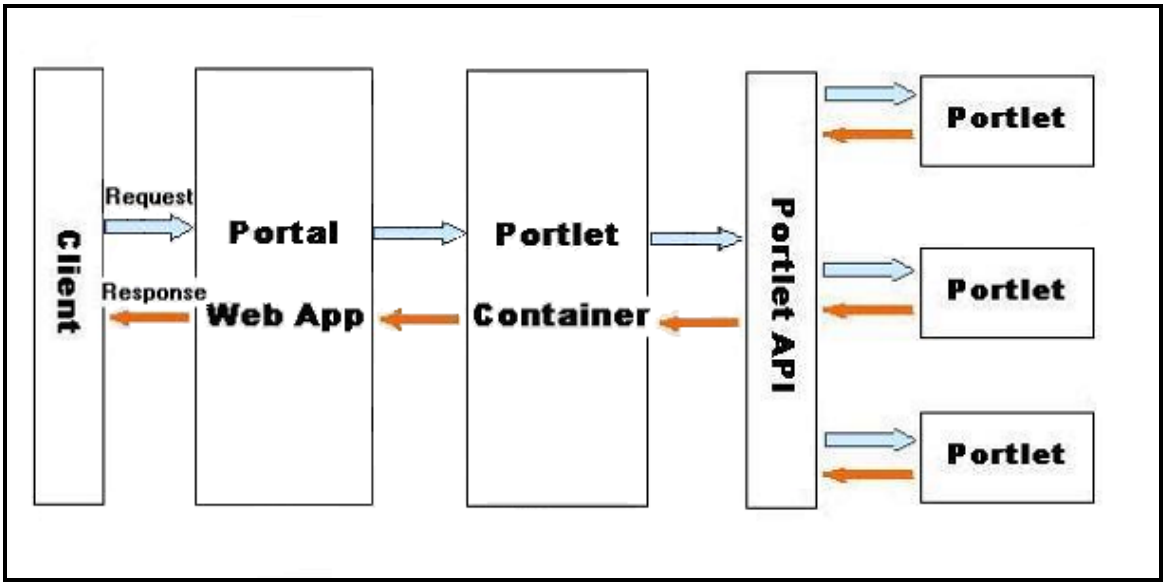

Figure 2.4: A typical scenario for the generation of a portal page.

2.2.1.3 Portlet lifecycle

A portlet container manages a portlet's lifecycle by invoking the corresponding methods defined in the Portlet [JSR168 2005] interface. These are described as follows:

- **init():** Invoked by the container to initialize a portlet. This method can be used to allocate resources required by a portlet.
- **render():** Used to generate portal page fragments.
- **processAction():** Indicates that a user has performed a specific action on the portal page, such as clicking a link.
- **destroy():** Indicates the end of a portlet's lifecycle. Usually, this method is used to free up resources allocated in the init() method.

2.2.1.4 Portlet Modes

A portlet mode indicates the function a portlet is performing. Normally, portlets perform different tasks and create different content depending on the function they are currently performing. A portlet mode advises the portlet what task it should perform and what content it should generate [JSR168 2005]. JSR 168 defines the following required modes. Portal vendors may include more vendor-specific modes in their own products.

- **View:** In this mode, a portlet usually renders static content such as text.
- **Edit:** In this mode, a portlet usually renders content that requires user interaction. This allows users to enter customized data.
- **Help:** In this mode, a portlet usually renders online help information for users.

A portlet container is responsible for providing the current portlet mode to the portlet. Upon receiving the current mode, a portlet render method is usually implemented to generate different content according to the mode. For example, the render method may display an editable input box on a page in the edit mode or a non-editable text in the view mode. Usually, the render method delegates this functionality to some other methods as below:

- doView(): invoked by render() when the portlet is in View mode.
- doEdit(): invoked by render() when the portlet is in Edit mode.
- doHelp(): invoked by render() when the portlet is in Help mode.

These are the interface methods required by and implemented in a portlet to interact with its container. WIT is implemented to test these methods.

The portlet modes that a portlet can support are configurable when the portlet is deployed into a portal. A portal is also responsible for rendering the portlet according to its mode on the portal page. For example, Figure 2.5 shows how a portlet configured with Edit and Help mode is rendered on a portal page. The portlet can be switched to Edit or Help mode by clicking the corresponding buttons.

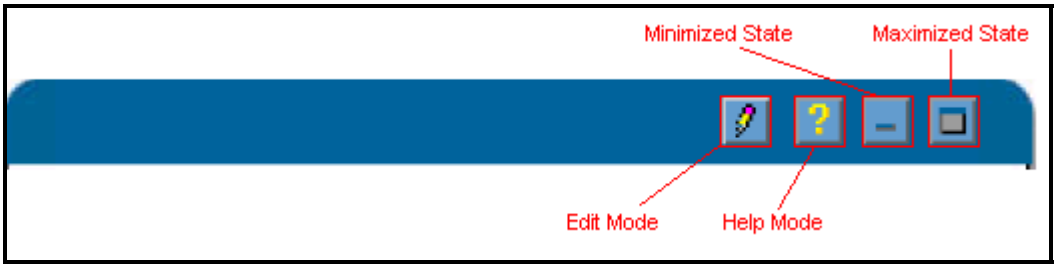

Figure 2.5: An example of portlet modes and states on portal page

2.2.1.5 Portlet States

Portlet states control the size of area where a portlet is rendered on a portal page. JSR 168 defines three states as below. Vendors can add their own states in addition.

- Normal: This state indicates that a portlet can occupy a certain visible area on the portal page.
- Maximized: This state indicates that a portlet is rendered in the entire body of portal page.
- Minimized: This state indicates that a portlet is invisible and thus should not be rendered on a portal page at all.

Similar to portlet modes, portlet states are rendered on a portal page by a portal. For example, Figure 2.5 shows a how a portlet configured with Minimized and Maximized states is rendered on a portal page.

2.2.2 IBM WebSphere Portal Server [WPS2005]

In addition to the common features described in 2.1, WPS provides the following extra features:

- **Content Management:** provides a centralized repository for documents. Document access control only allows authorized users to access documents. Document version control tracks changes and comments. In addition, WPS Document management enables remote users to edit documents even if they do not have editor software on their personal system.
- **Collaboration:** WPS provides collaboration services such as instant message, email, and chat room to help people work together and share information with each other.
- **Multi-device support:** In addition to desktop browsers, WPS applications are accessible to other devices, such as a cell phone, PDA. WPS is able to recognize the type of devices automatically, so it can adjust the page size and layout in terms of the device specifications.

2.2.2.1 Portlets in WPS

WPS version 5.0.2 or above is fully compliant with JSR 168. It provides a supporting environment for the deployment and execution of JSR 168 compliant portlets. However, WPS has its own portlets that is very similar to but still different from JSR 168 portlets. These portlets differ in aspects such as life cycle management, API definition, packaging and descriptor. Please refer to [IBM 2006] for more details. This thesis intends to address issues related to JSR 168 compliant portlets not WPS portlets, hence, the term ‘portlet’ in the following sections is used for JSR 168 compliant portlets.

2.3 Software Testing

This research addresses how to improve testing for web portal applications. To help readers understand how the research achieves this goal, the previous section 2.2 provides a basic background of the web portal technology. The purpose of this section is to give readers a brief introduction to software testing, with a concentration on web application testing tools.

2.3.1 Software Failures and Faults

Software failures are described as the variations in the execution of software code from the software requirements. For instance, the specification may include a security requirement that only authenticated and authorized users with appropriate permission can change the configuration of a Portlet page. If an anonymous user is able to do such change, a failure occurs. Such failures may be the result of either incorrect statement of the specification or the incorrect implementation of the specification, the software faults. The purpose of software testing is to find as many software faults in the designed and implemented code as possible, thus reducing the failures caused by them. Software faults usually are also known as software bugs. Bug fixing is the process of making changes to the existing code so that the detected bugs are removed.

2.3.2 Software Testing Process

A basic software testing process consists of three steps: 1. Set up an initial input state for the execution of software code. 2. Execute the code. 3. Check the output state generated by the execution of the software. An expected output state stands for a test success,

otherwise, it indicates a test failure has occurred. In the case of test failure, further analysis may be required to locate the exact location of the fault, thus it can be fixed later.

2.3.3 Categories of Software Testing

Software testing can be divided into several categories:

2.3.3.1 Unit Testing and Integration Testing

Usually, a large system is divided into several reusable components or modules, which can be further divided. Unit testing is a procedure used to validate that a particular module of source code is working properly [UnitTest 2006]. For instance, modules in applications written in the Java programming language are known as *methods*. The aim of unit testing in Java-based applications is to ensure that the execution of a method is correct according to the method design by checking the input and output data. Unit testing provides benefits such as improving software quality, reducing software maintenance cost, and improving software design.

When collections of modules have been unit-tested, integration testing is performed to verify the interfaces between modules. This ensures that the modules still work properly when they are assembled together.

2.3.3.2 Functional and Non-Functional Testing

Functional Testing ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions [IEEE 1990]. For instance, a functional requirement may need a portlet to display a monthly income report for employees. In this case, the functional testing can validate this

function by checking the generated report data for a specified employee. While functional testing ensures that a software system or module works, Non-Functional Testing verifies how well it works. Typical examples of non-functional testing are performance testing, security testing, and stress testing. In the scenario above, a non-functional requirement may need such a report to be rendered within 5 seconds. Non-functional testing verifies if the execution of portlet code meets such a performance goal.

2.3.3.3 Functional Unit Testing

Functional unit tests differ from functional tests in the granularity of the test level. Functional testing concentrates on verifying the functionalities of a software system or modules. For example, in a web portal, functional testing could monitor the intended behaviours of a portlet on a page, such as if the portlet renders correct content. Functional unit testing, on the other hand, validates the functionalities of a piece of software at the unit level. For example, this testing technique tests the behaviours of a portlet method when the portlet services to clients as part of functionalities of whole software. Thus, a functional unit test works at a finer level of granularity than the coarse-grained functional tests. When a failure occurs, functional unit tests help to narrow down the scope of pieces involved in the system.

2.3.3.4 Black Box testing and White Box Testing

Functional testing is also known as black-box testing [IEEE 90].

In such a testing technique, testers do not need to have programming knowledge. Instead, they only need to understand and examine the functions the code should provide.

On the other hand, white box testing examines the internal behaviours of a software system or module and the tests are prepared with explicit knowledge of how the internal code works.

2.3.3.5 Acceptance Testing and Installation Testing

Acceptance testing is the testing process conducted by customers who based on the results, either grant or refuse acceptance of the software system being tested. It is often referred to as beta testing. Such testing is performed according to the customer's understanding of the software system. After the acceptance testing, the system is installed in the production environment in which it will be used. Another acceptance testing may be required to ensure that the system still behaves properly in the new environment. Acceptance testing happening in the production environment is also called Installation Testing.

2.4 Summary

This chapter presents the technical background of my research. It first brings an overview of web portal technology, a special type of web technology. Next, it describes how this technology is applied in the Java world. JSR 168, a Java standard for portal technology, defines a component named portlet, which generates a piece of content on a portal page. JSR 168 also describes the underlying infrastructure for portlets, such as a portlet container and a portal. A portlet container manages the lifecycle of a portlet and responds the content generated by each portlet to portal, which in turn integrates these contents into a complete portal page.

This chapter also has a brief introduction to software testing. Software testing approaches can be divided into different categories, such as unit test, integration test, etc. The following Chapter 3 discusses several testing tools related to my research.

CHAPTER THREE: RELATED WORK

To prove the feasibility of in container testing, a tool, Web Portal In-Container Unit Testing Framework (WIT), was built. This chapter describes several existing Java testing tools related to WIT. It briefly presents how these tools are used to perform software testing, the benefits by using them, and the weakness when utilizing them to test web portal applications. WIT was developed to overcome the main issues identified in other tools.

3.1 Junit

[Junit 2005] is written by Erich Gamma and Kent Beck, and has become the de-facto standard tool for conducting unit tests in Java.

JUnit provides a base class called TestCase that can be extended to create a series of tests for the Java class under testing. Inside the subclass of JUnit TestCase, developers define any number of test methods that invoke the original methods of the class under testing. The actual test results are checked against the expected results by invoking assert() methods. Two of TestCase's methods, setUp() and tearDown() are used to initialize and release any common objects used for testing. Such objects are referred to as a *test fixture*. To ensure other methods are not affected, each test method runs within its own fixture. setUp() is invoked to set up the testing environment before the execution of test method, and tearDown() is used to restore the testing environment after the test is performed.

In a large application, there may be a huge number of test cases. To make test cases manageable, test cases can be composed into TestSuites that in turn can be a composition of other TestSuites. In this way, a tester is able to link test cases in a tree-like structure and run all the test cases in a TestSuite.

3.1.1 Advantages

Using JUnit, testing is closely integrated with the development process and test code is coupled tightly to the application code. This makes it possible to build test suites incrementally that help to focus on development efforts, identify errors as well as measure progress. Moreover, as JUnit is implemented in Java and available as an open source project it can be used and extended easily.

3.1.2 Weakness when testing portal applications

One aspect lacking in JUnit testing is a convenient way to test methods that need environment information provided by the server. This is limited by the absence of an API to control the required environment variables to test the methods that require input of the run-time environment. Setting up the system state for such methods under tests requires the ability to access and control the run-time environment. As a result, JUnit cannot independently test components such as portlets, because these components need the run time environment of the container in order to execute. For example, the doView() method in a portlet requires a PortletRequest object to be provided by a portlet container.

3.2 Cactus

[Cactus 2005] is an extension of JUnit and is designed for performing unit tests on in-container Java components, such as Servlets or EJBs. Cactus implements an in-container strategy, meaning that tests are executed inside the container [Cactus 2005].

3.2.1 How Cactus works

A Cactus test case is written in a similar style as JUnit except that there are three test methods in Cactus for each method under testing, begin(), test() and end().

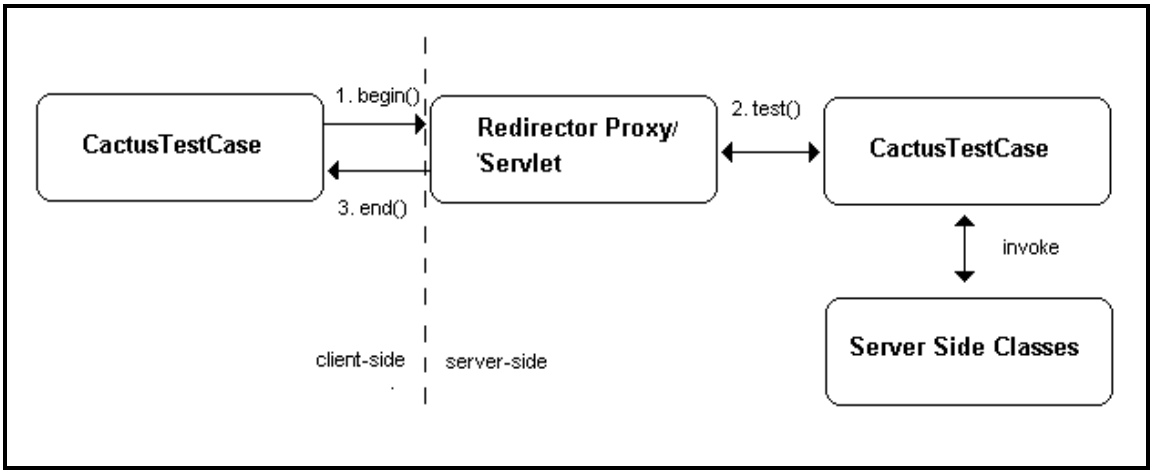

Figure 3.1: A simplified diagram that explains how Cactus works from [Cactus 2005].

As depicted above:

1. The begin() method opens an HTTP connection to the Redirector Proxy installed on a web server, and test data, such as the HTTP request parameters, are prepared at this time.
2. Upon receiving the test data from the client side, the Redirector Proxy forwards the test data to the test() method of the CactusTestCase. At this time, the Redirector

Proxy creates a new instance of the server side component class, and invokes the new instance's methods to be tested with prepared data.

3. The test results are captured by the Redirector Proxy and sent back to the client-side end() method where testers have the chance to review the response.

CactusTestCase defines several objects, which can be referred to in test code and are populated by Redirector Proxy in runtime, such as HttpServletRequest, HttpServletResponse, and HttpSession. Those objects are generated by a container and involved in the execution of server-side classes during testing. In other words, the server-side classes under testing are tested against test data contained in real container objects.

3.2.2 Advantages

In order to test application code that uses services provided by a real container the test code must be executed against the real container. Cactus supports executing unit tests of components (such as servlets, taglibs, JSPs), assuring that when the code is deployed in the production environment, the component logic will perform as expected.

3.2.3 Weakness when testing portal applications

Although Cactus test cases are run in a real container and thus can be used to perform in-container tests for components like servlets, some in-container methods and their interactions with the real container cannot be completely tested. This is described in further detail below.

First of all, the interactions between all server-side classes and the container are incomplete. A container interacts with server-side classes via the interface methods defined inside them. For instance, a container instantiates a Servlet component by

invoking its init() method along with an input parameter. The ServletConfig object, which is generated by the container, contains information like the initial parameter and the declared name of the servlet. When a service request for this Servlet is received, the container packages request data as an HttpServletRequest object and then passes it into the service() method of Servlet. When the container determines that a Servlet should be removed from service (for example, when a container wants to reclaim memory resources, or when it is being shut down), it calls the destroy() method of the Servlet. Cactus instantiates the server-side component classes directly and simulates container interactions by calling interface methods of the server-side classes. Even though Cactus is able to pass objects that are generated by the container into these interface methods, it does not act exactly the same way as a container. For example, the time to destroy a Servlet is determined by the container based on its own criteria. Even though Cactus can manually call the destroy method upon the servlet, the criteria to trigger such an action is unknown to it.

Further, container objects such as HttpServletRequest and HttpServletResponse passed by Cactus into interface methods are not exactly the ones that would be generated by a real container for server-side classes. Rather, these container objects belong to the Redirector Proxy, a Servlet component installed in a real container. As depicted in figure 3.2, a Redirector Proxy resides in a real servlet container. It is assigned all the required container objects and managed by the container. During the Cactus runtime, a Cactus test request is sent from the client side CactusTestCase to the container where the Proxy servlet sits. The container initializes the Proxy servlet by creating new instances of

container objects for it, such as HttpServletConfig, HttpServletRequest and HttpServletResponse. These container objects are created specifically for the Proxy; however, Cactus passes them to server side classes to validate the execution of server side classes in the container.

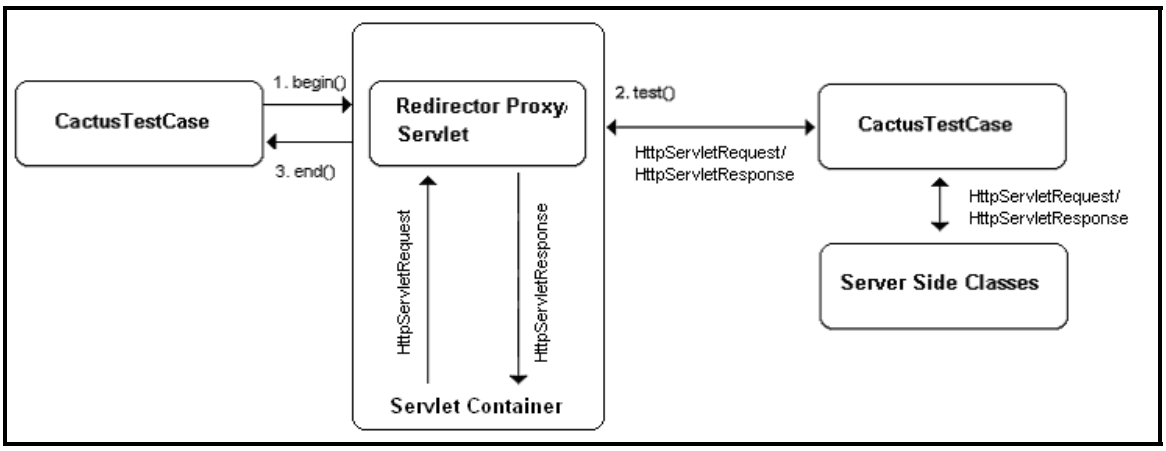

Figure 3.2: Cactus Proxy passes its own container objects to classes under testing

This kind of replacement may cause inadequate detection of deployment-related errors as well as security issues. For example, in a J2EE-based container, a servlet can acquire external information from parameters configured in a file, called deployment descriptor. At runtime, parameters configured specifically for the servlet are packaged into a ServletConfig object by a servlet container and passed to the servlet. Suppose, a developer has implemented a servlet method that calculates property mortgage rate based on the prime rate defined as an external parameter. Cactus would not able to properly test such a method because the ServletConfig parameter passed to the servlet method during testing belongs to the Redirector Proxy, which may not have been configured with the initial parameter.

3.3 Mock Object Approaches

A mock object is a substitute implementation to emulate or instrument the real code [MockObjects 2006]. The advantages and weakness of this technology are described as follows.

3.3.1 Advantages

Mock objects can be used to facilitate unit tests when the testing scenarios are impossible to prepare, or, if possible, perhaps too time-consuming. For example, to test how a Portlet reacts to a wrong database configuration, the Portal server has to be shutdown and reconfigured and then restarted. Tim Mackinnon [Hunt 2001] offers the following list to explain why a real object can be difficult to test:

- The real object has nondeterministic behavior (it produces unpredictable results; as in a stock-market quote feed.)
- The real object is difficult to set up.
- The real object has behavior that is hard to trigger (for example, a network error).
- The real object is slow.
- The real object has (or is) a user interface.
- The test needs to ask the real object about how it was used (for example, a test might need to check to see that a callback function was actually called).
- The real object does not yet exist (a common problem when interfacing with other teams or new hardware systems).

Using mock objects, it is possible to provide a workaround for all these problems. The essential idea of utilizing mock objects to perform a unit test is to reduce the dependencies of the code under testing on the dependent code. By providing a simpler simulation of the behaviors of dependent code, mock objects make the unit tests fast to run and independent of the real target environment.

Tim Mackinnon also brought out a template for unit testing in conjunction with mock objects. It is described as below.

- Create instances of Mock Objects
- Set the states in the Mock Objects
- Set expectations in the Mock Objects
- Invoke domain code with Mock Objects as parameters
- Verify consistency in the Mock Objects

In a large software system, tracing the root cause of a failure is difficult because the failure exposed to testers may be a consequence of nested unknown failures. With this template, a mock object checks the expected and actual results each time it interacts with real code. The test is then more likely to fail at the right time and generate a useful message. Furthermore, as the behavior of the mock object is tightly controlled, a failure is very likely coming from the code under test. In other words, such a pattern provides a finer level of granularity of testing because the tests are executed inside mock objects and happen when interacting with dependent code.

3.3.2 Weakness when testing portal applications

Literally, mock objects are not the real objects with which the code under testing is going to interact in a production environment. The simulation of a live environment makes the testing feasible; however, such a fake implementation of real objects cannot guarantee the validity of test results in the real production environment. When applied to a portlet application, mock objects need to be implemented to mimic portlet containers. In a production environment, however, this kind of container substitute may cause unexpected behaviors or even failures of components, because functionalities of components exposed to outside clients are really a combination of components and containers. In other words, such a testing approach is inadequate to capture failures that only surface in a real container. Rather, it concentrates on validating the correctness of a component's internal logic.

Another disadvantage comes from the implementation and maintenance cost of mock objects. To replace the real objects with mocked implementation, testers have to read and analyze the existing code, then implement mock objects based on their understanding of API specifications and the interactions between the API and the code to be unit tested. Under some circumstances, implementing mock objects may be very costly and the benefits from such unit testing approach are thus reduced.

Furthermore, there are cases where the real code cannot be replaced. For instance, calls to static methods are difficult to verify or replace because mock-object testing relies on the manual replacement of real code with test classes that share a common interface

specification. Because static method calls cannot be overridden, calls to them cannot be replaced in the way instance methods can.

3.3.3 Tools to support unit testing with mock objects

Currently, unit testing with mock objects plays an important role in the industry, especially in the areas like unit tests for databases, Servlets and EJBs.

- MockObjects

  [MockObjects 2006] is a generic unit-testing framework whose goal is to facilitate developing unit tests in the mock object style. It provides a set of mock implementation for the standard Java platform API packages such as servlets, sql, and io.

- MockMaker and EasyMock

  Both [MockMaker 2006] and [EasyMock 2006] are automatic mock objects generators, which reduce testers' workload by removing the manual implementation of mock objects. The difference between the two is that the former generates mock object source code at build time, while the latter directly generates those objects at run time.

- Mockrunner

  [MockRunner 2006] is a lightweight framework for unit testing applications in the J2EE environment. It supports Struts actions and forms, servlets, filters and tag classes. Furthermore it includes a JDBC test framework. The JDBC test framework can be used standalone or in conjunction with MockEJB to test EJB based applications.

- MockEJB

  [MockEJB 2006] is a lightweight framework for running EJBs outside of the container for unit testing. MockEJB allows developers to run EJBs directly from their favorite IDE with little setup effort. MockEJB is not a fully developed EJB container. MockEJB does not fully implement EJB or any other J2EE spec. It merely provides a convenient and realistic EJB testing environment using the mock objects approach.

- PortletUnit

  [PortletUnit 2006] is a JUnit Java Unit Testing Framework for testing JSR-168 Portlets. It is built on ServletUnit and Pluto. It provides a mock Portlet container as ServletUnit provides a mock servlet container.

3.4 Functional testing tools for Web Applications

A functional test evaluates the system to determine if the functions described by requirements specifications are actually performed by the integrated system [Shari Pfleeger]. Unit tests only ensure that a small piece of code functions correctly. No matter how well a software system is unit-tested, functional testing is still required to make sure the whole system behaves properly from an external client's point of view.

To perform functional testing for web applications, the most common approach is to simulate a browser's operations to interact with the Web server, and then analyze the responses received. Tools, like HttpUnit, jWebUnit and HtmlUnit, open an http connection to resources on a web server to initialize a test process. Upon receiving

response data from the server, tests are able to check the expected and actual data with such tools, such as the existence of page elements. The tests can even perform actions on the returned data, for example invoking a click action on a link element. Some other tools like [Watir 2006] drive a browser directly to communicate with the server.

Testing results of functional testing expose software system behaviours closest to what a real user will experience. A strong functional testing process provides developers with confidence before the system is delivered to customers. However, there are some limitations in utilizing this approach for portal applications.

3.4.1 Weakness when testing portal applications

First of all, constructing connections to a web server and obtaining responses back to the browser are time-consuming. A lengthy execution of a functional test means such a test technique will not be executed frequently especially in a resource-tight project. The value gained from such an approach is further reduced in a quickly changing environment because of its relatively slow execution compared to some other testing tools, such as JUnit.

On top of this, to validate the value of elements on the returned page, existing functional testing tools for web applications require a unique identification or name for these elements. For example, a web application displays a student's last name in a textbox, a page element that contains text data. Prior to checking if the last name was returned as expected, testing tools need a unique reference to the textbox in order to obtain the value. In WPS (IBM WebSphere Portal Server), however, this approach is not feasible because

the server dynamically generates a unique identification for elements. It is impossible for a developer to indicate to a testing tool that elements need to be checked.

Tracing the root cause of a failure is another issue with such tools. A failure observed on the browser side may involve a chain of invocations, from the database query to logic computation, or from the component to the container. Narrowing down the problem may be costly because of the needs to analyze each involved part.

3.5 Testing with AspectJ

To understand testing with AspectJ technology, we first present a brief introduction to aspect-oriented programming and its implementation in Java, AspectJ.

3.5.1 Aspect Oriented Programming (AOP)

“Aspect-oriented programming is a way of modularizing crosscutting concerns much like object-oriented programming (OOP) is a way of modularizing common concerns. For object-oriented programming languages, the natural unit of modularity is the class. But in object-oriented programming languages, crosscutting concerns are not easily turned into classes precisely because they cut across classes, and so these aren't reusable, they can't be refined or inherited, they are spread through out the program in an undisciplined way; in short, they are difficult to work with.” [AspectJ 2006] For example, a course registration application implemented in OOP can be modularized as a student and a course class. To implement a requirement that only an administrator user can modify student and course information, a developer writes code first to implement such logic and then modifies student and course class to refer to the new code. Obviously, such an

implementation where the new code referred multiple times increases software maintenance cost. For example, if the student class does not need the security check later on, the developer has to modify the code in the student class to remove the reference. Or if a new class is introduced into the system, it has to be modified to apply the security check. To provide a better solution for such issue, AOP is introduced to add or change functionalities such as adding the security check, without changing the existing source code in the system. In AOP, such distributed code discussed above is modularized as aspects.

3.5.2 AspectJ

[AspectJ 2006] is a seamless aspect-oriented extension to the Java programming language. AspectJ enables clean modularization of crosscutting concerns, such as error checking and handling, synchronization, context-sensitive behaviour, performance optimizations, monitoring and logging, debugging support, and multi-object protocols [AspectJ 2006].

In AspectJ, an aspect consists of pointcuts, advices and inter-type declarations.

- Pointcuts: indicate the places (joint point) in the execution of original code where the new code will be linked. For example, a method execution join point encompasses the execution of a method body. It includes all the actions that comprise the execution of a method body, starting after all arguments are evaluated up to and including return (either normally or by throwing an exception). [AspectJ 2006]

- Advices: is the code to be added into original code. A ‘Before advice’ runs as a join point is reached, before the program proceeds with the join point. For example, ‘before advice’ on a method execution join point runs before the actual method body starts running, just after the arguments to the method call are evaluated. “After advice” on a particular join point runs after the program proceeds with that join point. For example, the ‘after advice’ on a method execution join point runs after the method body has run, just before control is returned to the caller. [AspectJ 2006]
- Inter-type declarations: are declarations that cut across classes and their hierarchies. They may declare members that cut across multiple classes, or change the inheritance relationship between classes. New class instance variables or methods can be introduced to an existing class without changing the declaration of class with this feature. [AspectJ 2006]

3.5.3 Testing with AspectJ

Due to its ability to handle crosscutting concerns, AspectJ is helpful in performing test in areas where traditional testing tools are inadequate.

[Nicholas Lesiecki] proposed a solution for data-dependent testing with a combination of AspectJ and Mock object technology. In his article, Mock object tests replace domain dependencies with mock implementations used only for testing. Mock objects, however, cannot help with global resources because mock-object testing relies on the manual replacement of domain classes with test classes that share a common interface. For

example, static method calls (and other types of global resource access) cannot be overridden, calls to them cannot be ‘redirected’ the way instance methods can. Another example: in a J2EE system, the means of getting an EJB instance is to look up a factory (the Home interface) from a JNDI context, and then call a creation method on the factory. The EJB’s JNDI lookup method is distributed in every place where an EJB instance is required. To replace EJBs with mock objects, all lookup occurrences may be checked and modified. AspectJ offers a workaround for these types of issues by cross cutting the structure of the tested code. In detail, it provides ‘pointcuts’ to locate the distributed occurrences of access to global resources, and ‘advices’ to replace the real code accessing global resources with mock objects.

[Wes Isberg] points out “AspectJ complements testing techniques and tools…” First of all, to avoid writing test code, he suggested utilizing the AspectJ language to specify and verify program invariants, code standards or specifications, which cross cut the whole system. For example, the following AspectJ aspect can be used to ensure that a non-public field can only be modified via a setter method. Whenever a violation occurs, it reports a warning message at compilation time.

```
declare warning : within(com.xerox..*) && set(!public * *)
      && !withincode(* set*(..)) :
   "writing field outside setter" ;
```

In addition, the author mentioned that AspectJ helped improve complicated integration tests, which involve multiple components, and thus cross cuts the system.

[PatternTesting 2005] is a testing framework that automatically verifies that Architecture/Design/Best practices recommendations are implemented correctly in the code. Aspects ensuring typical recommendations have been implemented, such as the one verifying that all calls to the database go through a JdbcDataAccess class and that none use JDBC directly.

3.5.4 Weakness when testing portal applications

The major issue when performing tests with AspectJ technology is the complexity of the language. Being a new language that is different from the traditional object-orient languages, it is difficult to learn.

Second, AspectJ by itself is not adequate to perform tests. It has to integrate with other testing tools to accomplish a complete test.

3.6 Reflection

[Reflection 2006] is an approach utilized by JUnit to test private fields and methods. Using the facilities of the class java.lang.Class and the Reflection API, programmers can gather information about classes in a JVM at runtime, including references to fields, methods and constructors. After obtaining these references in the form of Field, Method and Constructor objects, developers can manipulate them in ways such as creating an instance via the newInstance() method of Constructor object, invoking methods with parameters via the invoke() method of Method object and reading the value of fields via the get() method of Field object. However, there are some limitations when using Reflection to access private members.

- Security Issue

To access private members at runtime, developers have to invoke setAccessible(true) to inform the Security Manager that the reflected object should suppress Java language access checking when it is used. “The Security Manager is a class that allows applications to implement a security policy. It allows an application to determine, before performing a possibly unsafe or sensitive operation, what the operation is and whether it is being attempted in a security context that allows the operation to be performed” [J2SE 2006]. In a secured context like a production environment, the Security Manager in WPS server blocks attempts to access the private members.

- Strict Typed-Parameters

In Reflection, “methods are looked up with the given name and formal parameters of exactly the type developers indicate. In many cases, this suffices because developers often know the exact signature of the desired method in advance. However, this type may not be the exact type in the signature of the method in runtime, but instead may be a subtype or a type that is otherwise compatible for reflective invocation” [Holser 2006]. Such a limitation imposes a strict method signature definition on a Portlet. It also causes unexpected runtime exception if a method under testing accepts a parameter that is subtype of the defined type.

3.7 Summary

This chapter depicts several existing Java testing tools. For example, JUnit is used to test Java methods, and Cactus is used to test in container components like servlets. The Mock Object approach is useful when the test scenarios are hard to prepare or time-consuming. HttpUnit can be used to perform functional testing for web applications. AspectJ helps

testing by isolating the test code from production code. However, all these tools have their own weakness when testing web portal applications. Chapter 4 and 5 present a unique testing tool that can be used to perform such tests. The effect of this tool is evaluated in Chapter 6.

CHAPTER FOUR: WIT TOOL REQUIREMENTS AND DESIGN

To our best knowledge, our Web Portal In-container Unit Test Framework (WIT) is the only framework at this time that supports in-container testing of portlet-based applications. Some other testing tools like Cactus can support in-container testing for Servlets, EJBs and JSPs. However, they are not adapted to handle the challenges of testing portlet-based applications. Their weaknesses in fulfilling such testing are described in the previous sections.

4.1 Research Partner's Requirements

Initially, the requirements for the tool arose from a company, which provides technical consulting services for building applications on IBM WebSphere Portal Server (WPS). The company reported that:

1) Errors occurred when tested applications are installed in the production environment and

2) Manually testing the access to sensitive portal pages is very time-consuming.

4.1.1 Errors occurred when applications are installed in the production environment

The company adopts several techniques to ensure the correctness of the applications they build. They use such as Test Driven Development (TDD), which requires developers build test cases before they start to write application code. Tools, such as JUnit, are used to write and run unit test code. Upon the success of unit tests, applications are installed in a test environment to conduct an acceptance test, which requires that the application code run in a real container. After being fully tested in the test environment, applications are

finally installed in the production environment, which renders live services to customers. The major difference between a test environment and a production environment is that the latter has a higher performance hardware platform. In addition, databases in the test environment are populated with mock data and thus different from the real data in the production environment. The company tries to keep all supporting software in both environments consistent, such as the operating system version, etc.

Unpredictable errors were encountered when fully tested applications were installed in the production environment. On one hand, those errors need to be fixed as quickly as possible since it is critical to provide correct and robust services to customers. On the other hand, fixing such errors usually is very time-consuming. Developers have to reproduce errors in the test environment in order to figure out the root cause. Unfortunately, in most cases, those errors cannot be reproduced in the test environment. Checking through the huge volume of application logs of the production environment becomes the most effective approach and must be used even though it is extremely time-consuming.

In the following section we briefly discuss some of the problems encountered when an application is deployed in the production environment.
According to our analysis, those problems are often caused by unpredictable changes in the container environment.
a) Deployment related problems

Application deployment deals with how to make an application run in a specific environment. Besides the installation of the application code itself, an application deployment usually involves the configuration of supportive parameters. For WPS-based applications, configuration parameters are prepared and stored in files called deployment descriptor files. Deployment descriptor files are then read at deployment time and the container is configured accordingly. For example, a database connection string describes all the required information for a Portlet to access data from the database, such as the location, name, etc. At deployment time, a WPS server reads the connection strings and establishes connections to specified back-end databases. A WPS sever also is able to automatically manage database connections, such as automatically disconnecting a connection when necessary in order to release occupied resources. Such a technique has two benefits. First, it eases the burden of a Portlet developer, who is now able to concentrate on how to fulfill more valuable business requirements. Second, it increases the portability of a Portlet. Without hard-coding a connection string in code, a Portlet can be configured to collaborate with any databases.

In spite of the benefits described above, deployment descriptor files might cause issues to Portlet developers. For example, a Portlet configured with the connection string to database A in the testing environment, for some reason, is assigned a connection string to database B when deployed to the production environment. Portlet code executing successfully in the test environment may fail because of the changed connection string.

In addition to deployment descriptor files, to make a WPS-based application run in the production environment, library files, on which the application relies, should be introduced to the new environment as well. Another possible difference between the test and the production environment may be due to the fact that a different version of a library file is being referenced by the application code. Subtle differences between versions may cause the failure of a Portlet. For example, in a test environment, a Portlet developer may use the [LazyValidatorForm 2006] class to encapsulate user-typed data from Portal pages. The LazyValidatorForm class is first introduced in the [Struts 2006] library version 1.2.6. Apparently, a production environment configured with an old version of the library is not able to execute the Portlet code. The above examples highlight that the successful execution of the Portlet code in a test environment cannot guarantee its success in the production environment.

b) Problems arise from the interaction between a container and a Portlet in the form of request, response objects and other container objects. In a container-based WPS application, a Portlet container packages data from a browser as a request object. The object is then forwarded to the Portlet code through the container API. After the execution of the Portlet code, results generated are sent back to the browser as a response object, which is also assembled by the container. The request and response objects are primarily responsible for carrying the data back and forth between containers and Portlets. In addition, a WPS container provides several other container objects, such as the PortletConfig object, which contains configuration information from a deployment descriptor. A Portlet may rely on these container objects, especially on request and

response objects, to accomplish its business logic. Changes happening inside container objects might cause failures on a Portlet. Automatically testing a Portlet that relies on these objects requires a mechanism that allows developers to manipulate these objects. However, changing container objects might bring impacts on the other parts of an application, especially when container objects are sharable among Portlets. Therefore, an ideal testing approach should have no interference with the other parts of an application.

4.1.2 Role based testing of resource access.

A Portlet generates a fragment of a Portal page, which may be shown as an [HTML 2006] page in a user's browser window. Most of the company's clients have security requirements for accessing the data they publish through WPS-based applications. For example, a Portlet rendering employee salary reports should be only viewed by the employee him/herself or HR administrators. If an employee's salary is viewable to other employees, it indicates a security breach. WPS provides a set of mechanisms to fulfill various security requirements. Sensitive Portlets are protected by means of assigning roles to individual users or user groups. Each role stands for a group of access permissions to resources. Provided with a user management console, a WPS administrator can easily establish a security configuration for an application. Nevertheless, testing if the security is configured correctly or not has become a big burden for the company. Without automated testing tool support, the administrator setting the permissions must log in as a user with a specific role and test manually each time the applications are deployed in the production environment to verify whether the permissions have been correctly assigned.

4.2 In-Container Functional Testing Requirements

Chapter 3 describes how the company failed to find an appropriate solution from existing testing tools. Based on the analysis of the problems and the component-container technology, we find: 1. These kinds of problems are very generic in component-based applications, 2. These problems are all related to the interactions between a container and a component. Thus, a general approach that can be used to test the interactions between a container and a component is feasible to solve these problems. We believe that in-container functional testing (ICT) is such an approach. To prove the feasibility of the approach and solve the problems that the company reported, a new testing tool that supports ICT is required. ICT performs functional testing when a component runs in its container. It requires a tool to be able to:

- Functional test portlets
- Test portlet at the method level.

4.3 WIT Design Objectives

According to these two requirements identified above, WIT's design should address the following issues:

- Accessing and controlling container managed API objects.
- Executing test code in containers
- Testing portlet methods
- Testing access to portlets

- Minimizing test execution side effects
- Script supported tests

4.3.1 Accessing and controlling container managed API objects.

WPS defines a set of API objects through which containers expose their services to Portlets running inside of them. For example, a portlet acquires the database connection string configured in the deployment descriptor through a PortletConfig object. In other words, a Portlet cannot acquire underlying services such as a database connection directly without interacting with those API objects. Only when the test code is able to access and further interact with those objects can a test code set up the container environment according to a testing scenario, and verify the test results by analyzing the container environment after the test execution.

WPS container specific API objects are instantiated and populated in following scenarios:

- When a Portlet is deployed into a container. This is a relatively static scenario. A container would collect information from deployment descriptor files to create corresponding container objects. For example, an administrator can configure a currency rate for a Portlet which functions as a currency converter by designating the rate as a context parameter in the deployment descriptor file. After the Portlet is deployed, the rate information is available to the Portlet via a PortletConfig object.

- When a Portlet is running in container. Dynamic information like a request from a browser is collected by the container and provided to a Portlet as a container API object. For example, a PortletRequest object encapsulates information submitted from a client browser.

The WIT design should support the access and control of container specific API objects involved in both scenarios.

4.3.2 Executing test code in containers

Executing test code in containers consists of three parts. First, it requires that test code is able to be deployed into containers; Second, it requires that test code is able to be triggered and thus executed in containers; Third, it requires that test results is able to be collected in containers.

4.3.3 Testing Public and Private Portlet Methods.

Functional unit testing requires a tool to test portlet methods, public or non-public. Testing non-public methods is an approach bearing ongoing debates. Many argue that only the public methods should be tested since it is only via public methods that functionality of code is exposed to an outside invoker. Private methods should be protected by the test coverage of public methods, which are being tested. On the other hand, non-public methods can contain complicated business logic and directly testing such methods can increase confidence in the test results.

The main purpose behind the design of a testing tool like WIT is to help developers to minimize the scope of problems in a WPS environment. It is still arguable to test non-public methods, however, in our opinion, the approach helps to locate the source of a problem and increases the benefits the tool can provide. Even if developers think that non-public methods should not be tested, having the option handy provides them the chance to try both testing approaches and then determine how to apply a better testing strategy.

4.3.4 Testing access to portlets

A software system built on WPS faces the same security challenges as the other web-based applications, such as how to protect important data from being accessed by invalid users, how to make sure data is transferred without being maliciously tampered. A tool that intends to help enforce security for such an application should address, but not be limited to, the following security requirements.

- Conversation encryption

  To prevent sensitive data from being intercepted as it travels over the network, conversation between browsers and Web servers must be encrypted. Secure Sockets Layer (SSL) has become the most common method for creating an encrypted connection between clients and servers.

- Authentication

  Authentication is a process by which a system verifies that users have valid access credentials. Basic authentication methods include user ID and password, SSL certificates exchanged between client and server, etc. Authentication is equivalent to persons presenting their drivers license at a bank counter.

- Authorization

  Authorization is the process by which it is determined if an authenticated user has appropriate access permissions to sensitive resources such as Portlets. WebSphere Portal Server grants permissions to users via a role type that groups a set of permissions together.

Security testing is a kind of non-functional test approach usually accomplished in a production environment. Manual security checking is costly and error-prone. At the time of writing, the current design of WIT only focuses on the authorization part of security tests on Portlets. For example, it can report a failure to developers if a user without view permission of a Portlet is able to browse the content rendered by the Portlet.

4.3.5 Minimizing test execution side effects

The execution of in-container test code may impact the original Portlet code in two ways. First, container provided API objects are accessible to Portlet code and the testing code. Such access sharing brings conflicts when Portlet code and testing code try to access and control the same API object. For example, a Portlet which displays the detail list for an invoice retrieves the invoice detail information from a database according to the invoice number sent from a browser. In WPS, the invoice number is packaged into a PortletRequest object by the container. Since the test code has the ability of modifying container generated objects, the original value of the invoice number possibly has been altered before it reaches the Portlet code. As a result, a wrong invoice detail list is rendered in user's browser window. Also, an unexpected change to container objects

might also cause a test result to be invalid. The WIT design should eliminate such access conflicts to make sure a Portlet functions properly while it is being tested.

Secondly, the performance of the original portlet code may be reduced as the test code sits inside of portlet code and intercepts the calls to it. WIT design should minimize such a side effects as much as possible.

4.3.6 Scripts for testing.

Most of the existing Java-based testing tools are using test scripts to organize individual test cases together. With a test script, test cases are more manageable and understandable. Furthermore, test scripts make it easy to perform regression testing as they provide developers with a centralized place to execute all test cases. Regression testing is a re-testing of a modified software application to ensure that the application continues to function correctly as before. The WIT design should support script-based testing in order to be integrated with other popular testing tools to augment the value of the tool.

4.4 WIT Design Challenges

With commercial portal servers, the source code of the portal server components as highlighted in Figure 4.1 is inaccessible to the developer.

Because of the complexity of web portals, automated in-container testing presents four unique challenges.

Firstly, the portlet API layer depicted in Figure 4.1 is the only way that portlets can ‘talk’ to the inaccessible components. Thus, we need to find a way to intercept calls from the

container to the portlets and vice versa so that testers can access and manipulate the calls generated by the container.

Secondly, testing portlets involves invoking a series of inaccessible interactions in the portal server as seen in the Figure 4.1.

Thirdly, since the tests run in the container we need to collect individual test results of executing each portlet test and then send back the aggregated results to the test client.

Lastly, while the test code runs with the original application code, portal clients still should receive the correct response from the portlets. Thus, minimizing the side effects of the test code on the original portlet code becomes essential.

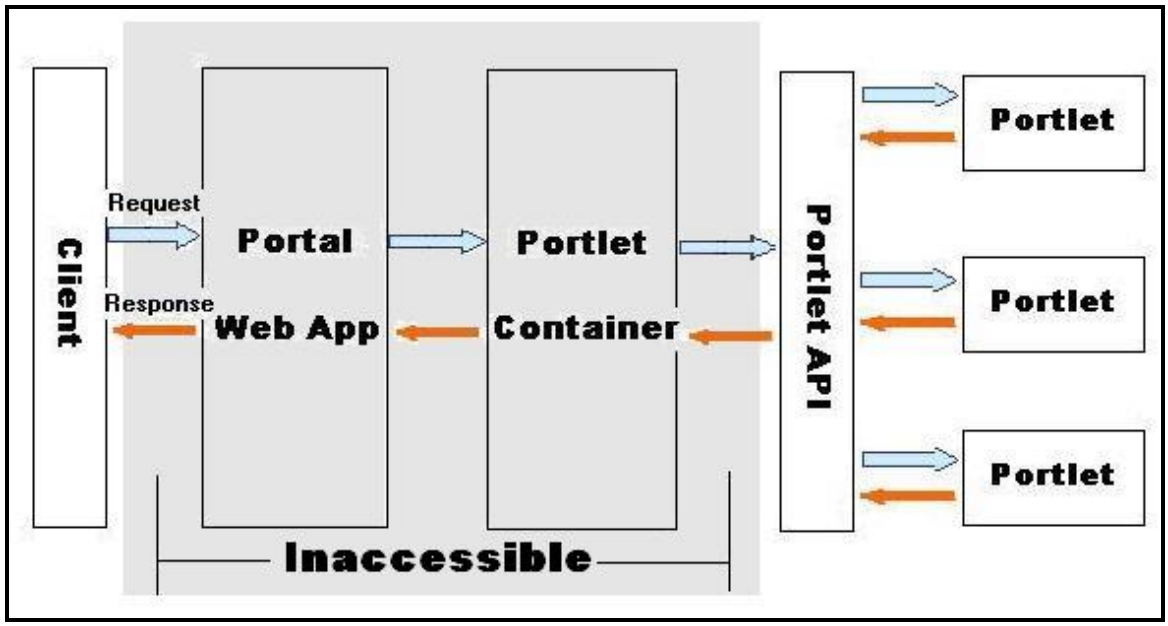

Figure 4.1: Inaccessible Portal and Portlet Container

4.5 WIT Design

This section presents the design details of WIT. WIT consists of several modules. They are Converter, Weaver, Invoker, Controller and Repository. The system architecture of WIT is depicted in Figure 4.2.

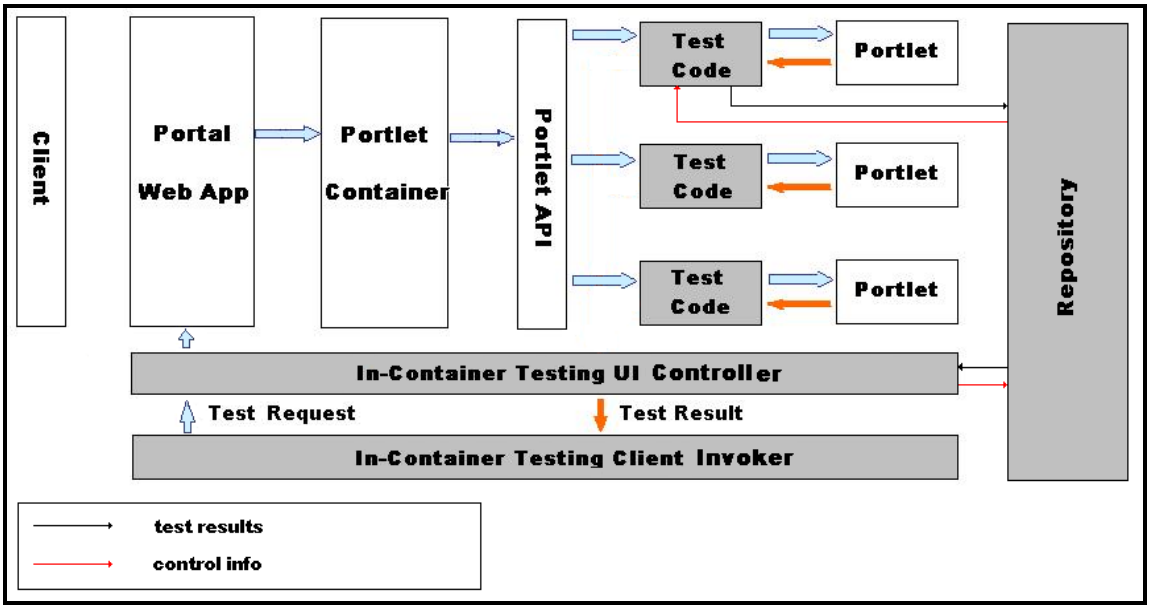

Figure 4.2: WIT architecture

The Testing Client Invoker depicted in Figure 4.2 initiates tests by establishing an HTTP connection and sending a test request to the Controller. This starts a process whereby the Controller forwards the request to the Portlet container that in turn forwards the request to Portlet under test. Meanwhile, the Controller writes a test control instruction to the

Repository. Before the request reaches the Portlet under testing, a check is made to ensure that the test control instruction allows the test code to execute. If the check is successful, the test code then:

1) Intercepts the calls between the Portlet code and Portlet API and

2) Sets up the initial state to execute the Portlet code.

3) Upon the completion of Portlet execution, the Portlet generates a response. Before the response is sent back to the container, another check is made by the test code to see if the information contained in the response is expected. The results of the check are recorded in the Repository. Later, the Controller collects test results from the Repository and then generates a report for the testers.

4.5.1 Invoker

The Invoker is the starting point of the in container testing process. The responsibilities of the Invoker are multifold. First, it is responsible for the preparation of the testing process by:

- Reading Configuration files which define information such as the definition of test suites, the URL indicating the location of the Controller, etc.
- Assembling the definition of test suites into an HTTP string parameter.
- Invoking the Converter as explained in Figure 4.3 to wrap Portlet code with test code and then deploys wrapped Portlet code into the target Portal server.

When all the above steps are finished, it establishes an HTTP connection with the Controller and sends out a test request along with test suite information. The Invoker is

also responsible for receiving and formatting the response from the Controller and then generating test reports for developers. For production environment WPS servers, which are secured in a separate room or building, usually have no local access to testers, the design described as above is useful for deploying tests on a production server and perform the test remotely via an Internet connection.

4.5.2 Controller

The Controller is a servlet deployed in the WPS container to accept the test request from the Invoker. It parses the test request string to obtain the URL to which the Portlet under test is assigned, and then simulates the invocation of the Portlet from a browser by sending an HTTP request to the URL. The WPS server then assembles environment objects required by the execution of Portlet, such as the PortletRequest object. Before the Controller sends out the HTTP request to a Portlet URL, it writes a control instruction into the Repository to indicate which method of which Portlet is going to be tested. Only the test code residing in the specified Portlet is activated upon receiving a test request.

The Controller is also responsible for querying the test results saved in the Repository and then sending them back to the Invoker.

4.5.2.1 Portlet URL and Login URL

Unlike Servlets, Portlets are not directly bound to a URL. To trigger the execution of a Portlet, WIT utilizes the URL to the page on which the Portlet is deployed. Upon receiving the page URL, WPS parses the page and then requests the Portlet container to

invoke all the Portlets deployed on the page. For convenience, this thesis considers a page URL a Portlet URL that stands for the URL to invoke a Portlet.

Portlets deployed in the production environment usually require user authentication and authorization. A user accomplishes the authentication via a login page with username and password. Before sending out the request to Portlet URL, the WIT Controller is able to send an authentication request to the login URL.

4.5.2.2 Criteria to determine a Portlet has not been tested

Only when a Portlet is able to react upon receiving a test request, can it be able to record its execution result into the Repository. Under some circumstances, Portlets to be tested may have no response. For example, the Portlet URL is incorrect, thus the test request cannot reach the Portlet. Another example is when the Portlets are un-deployed from the Portal server.

WIT defines two parameters to determine if a Portlet has not been invoked. AvgExeTime is the average execution time for each Portlet and RetryTimes is the maximal retry times for each Portlet. When AvgExeTime expires before the Portlet writes its execution result to the Repository, the Controller sends another test request to the same Portlet up to RetryTimes. At the end of RetryTimes, the Controller generates a non-execution report back to the Invoker if the execution result for the Portlet is still unavailable in the Repository.

4.5.3 Converter & Weaver

In order to intercept container calls to the Portlet code under test, and thus to obtain environment objects provided by the container, the Portlet code is wrapped by testing code in WIT. To do that, we utilized AspectJ technology to inject the test code into the Portlet code.

4.5.3.1 How WIT utilizes AspectJ to wrap Portlet code with test code

WIT test code does not need to be a distributed code, however AspectJ is still helpful when wrapping portlet code. As explained above, AspectJ pointcuts can be used to locate the methods of classes under test. A ‘before advice’ on a method execution can be used to intercept container environment objects passed to Portlet code, and thus to provide the tester a chance to manipulate those environment objects to set up a desired test environment. Following the execution of Portlet code, an ‘after advice’ can be used to validate the testing results caused by the execution of Portlet. The ‘after advices’ also provide testers a chance to clean up the testing environment so that the trace of in-container testing can be removed and the impact to the environment is minimized. Inter-type declarations can be used to introduce new methods and fields into portlet classes.

With WIT, as shown in Figure 4.3, test cases written for in-container testing are fed into the Converter first to generate Aspect code, which in turn is woven into the original Portlet code by the AspectJ Weaver. During this phase, auxiliary information like the location of the Repository is compiled into the Portlet code as well.

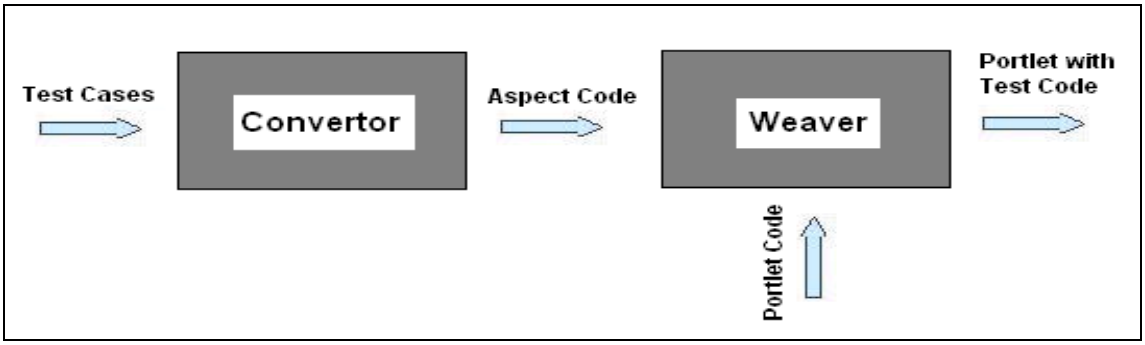

Figure 4.3: WIT Converter and Weaver

4.5.3.2 Benefits from utilizing AspectJ

First of all, AspectJ makes it possible to wrap Portlet code with WIT test code, and thus enables the test code to access and control the container-generated objects. It is the foundation of our implementation of an in-container testing approach. Granted with the ability of controlling the container environment objects, a tester is able to capture and customize the interactions between a Portlet and its container.

Thanks to AspectJ technology, WIT test code is separated from Portlet code. Such separation brings two benefits. First, the separated test code is easier to maintain. Test code can be placed in different locations and organized with different package namespaces. Second, the impact caused by test code on Portlets is reduced. AspectJ's weaver acts as a switch when wrapping the Portlet code. Test code can be woven into Portlets when performing in-container test. When the test is finished, test code can be peeled off from the portlets.

AspectJ also helps circumvent the restrictions of accessing private fields and methods of a class. Inter-type declarations are used to introduce test methods and fields into the

Portlet classes. With such an approach, private fields and methods are not longer ‘private’ for testing code.

4.5.3.3 WIT Test Case

AspectJ plays an important role in our in-container test implementation. Learning and mastering this technology is not an easy work. In order to make WIT more user-friendly, WIT defines a JUnit-styled test case while utilizing AspectJ technology behind the scenes. A tester is able to perform an in-container test without knowing the underlying AspectJ technical details. Details of a WIT test case are described in the following chapter.

4.5.4 Repository

The Repository is a central place where test results and control information are stored. A test procedure can be divided into two categories, synchronous and asynchronous. JUnit is the most typical example of synchronous procedure. A synchronous test procedure consists of a sequence of test iterations each of which starts from the invocation of test code to setup the testing environment. Following the test environment setup, the test code invokes code under testing and waits for the response. Test code then compares the response with expected output. A synchronous testing procedure ends up sending comparison data back to users. In a synchronous test procedure, users can obtain the test results immediately.

This approach, however, has performance issue when performing in container testing for Portlets. The execution of a Portlet might involve time-consuming operations such as

access to database. Obtaining the result of such an execution requires a long waiting period.

In contrast, in an asynchronous procedure, test cases are executed in parallel rather than one after the other. Test results generated in each execution are not sent back to users right away, instead, they are saved into a centralized repository. Later on, the results from the repository can be collected, formatted in the appropriate way, then sent back to users.

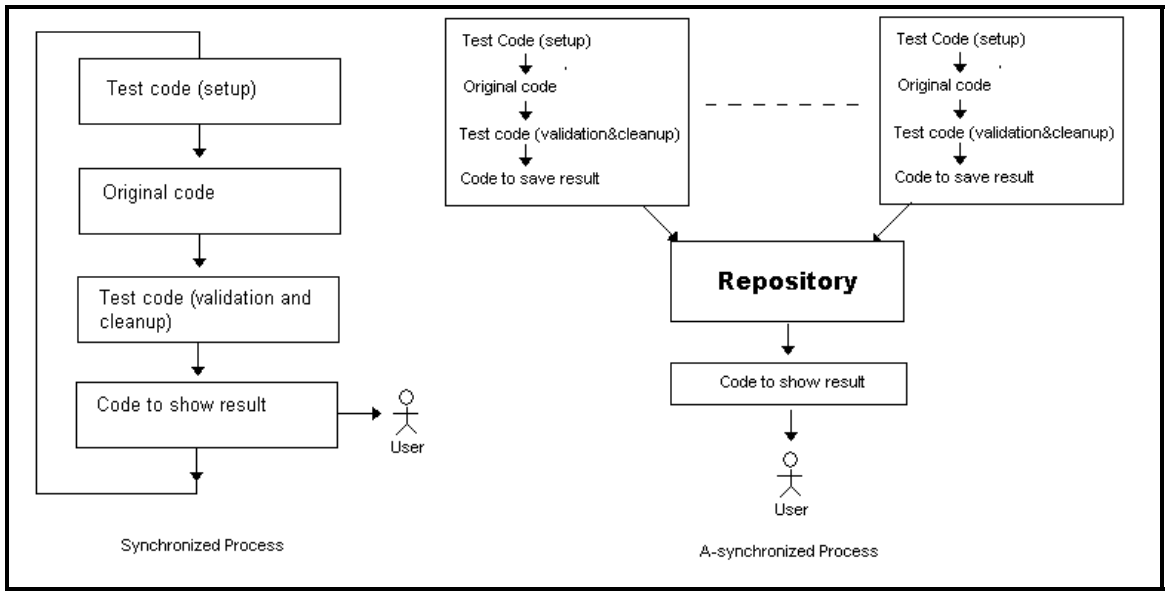

Figure 4.4: Synchronized Process VS. Asynchronized Process

In addition to contain test results for each Portlet, the Repository is used for exchanging control instructions between the Controller and the WIT test code. In a production environment, the requests for a Portlet are most likely from users' normal HTML page browse operation. In this scenario, the execution of test code embedded in a Portlet may interfere with the normal behaviour of Portlet. On the other hand, test code is required to be executed in order to perform an in-container test. How to distinguish the testing

requests from normal service requests is another issue when performing in-container Portlet testing.

In WIT, that issue is solved by a combination of embedding a unique global token string in the HTTP request object and placing control instructions in the centralized repository before the execution of test code. The unique token string indicates that the request comes from a test client. The token string has a server-wide effect; it turns ‘on’ or ‘off’ all test code that resides on the same server. More specifically, a control instruction indicates which Portlets are going to be tested. Only test code in Portlets specified by the control instruction is activated upon receiving a test request. In other cases, for instance, when a request from a real user reaches a Portlet, the embedded test code keeps silent, and the Portlet provides service to its client as usual. Such design works even in the case where a test request and a normal request are sent out for the same Portlet simultaneously, as WPS is able to instantiate multiple Portlet instances, one for the each request.

In order to provide better performance by avoiding I/O disk operations, we have chosen an in-memory database as our repository. [HSQLDB 2006] is the leading SQL relational database engine written in Java. It offers in-memory database tables and can be embedded into other applications. When multiple Portlet test code updates the same table simultaneously, access control is delegated to HSQLDB. WIT Controller is responsible for initiating the in-memory database during the WPS start-up phase and placing control instructions in it before the execution of test code. WIT Converter & Weaver injects API methods of the in-memory database into test code, through which the test code is able to acquire control instructions and save test results.

CHAPTER FIVE: USAGE SCENARIOS OF WIT

After providing an overview on the WIT architecture, we will now describe how a tester can write WIT in-container test cases. Three main usage scenarios of WIT will be discussed in detail with reference to an Accounts Portlet example.

5.1 Accounts Portlet

In IBM's Shipping Demo application, the Accounts Portlet is developed to display account information for a specified order. As shown in Figure 5.1, a user first types in an order number, and then clicks submit button. Detail account information shows up in the Figure 5.2.

Figure 5.1: Type in order id to show account details

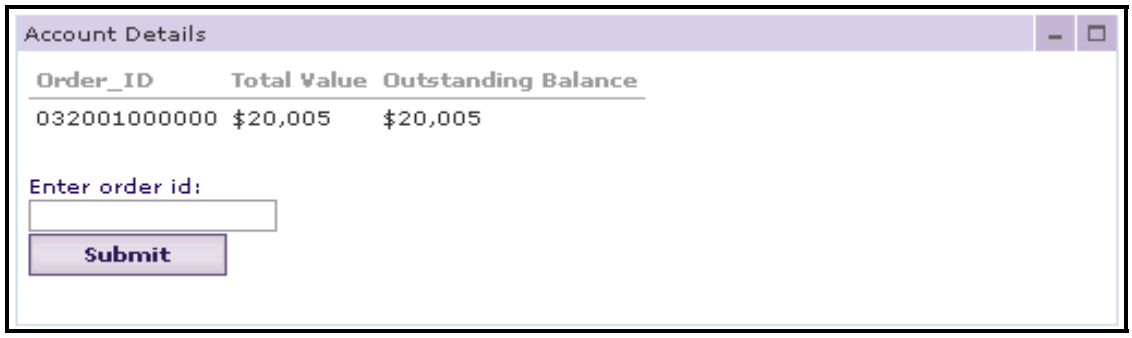

Figure 5.2: Account details

Figure 5.3 shows the source code of the Accounts Portlet class which contains a method called doView [JSR168 2005]. The doView() method is invoked by the Portlet container to generate mark-up fragments when the Portlet is in VIEW mode, the mode in which a user has a view-only access.

The Portlet container encapsulates all information about the client request, parameters, etc into a PortletRequest object, and all information to be returned from a Portlet to client into a PortletResponse object. In line 5, the Portlet accesses the JDBC database connection string to connect to the backend database. The corresponding order id is retrieved from the PortletSession object in line 7 and 8, which is sent to the back-end database system to get the detail account information packaged in an AccountDetail object in line 9. Line 10 shows that the acquired account data is saved in the PortletRequest object as an attribute, which is in turn returned to the client.

```
1)  public class AccountsPortlet extends PortletAdapter {
2)  public void doView(PortletRequest request, PortletResponse response){
3)  try {
4)  PortletSettings PortletSettings = request.getPortletSettings();
5)  String dbConnStr = PortletSettings.getAttribute("AccountDB");
6)  //Now the AccountsPortlet can persist information to the back-end Account database
     ……
7)  String orderId =
8)  (String)request.getPortletSession().getAttribute("orderId");
9)  AccountDetail ad = AccountDB.getAccountDetail(orderId);
10) Request.setAttribute(“AcctDtl”, ad);
11) PrintWriter out = response.getWriter();
12) //following pseudo code prints out the AccountDetail object
13) response.setContentType("text/html");
14) out.println(……);
15) }
```

Figure 5.3. doView Method – AccountsPortlet Class

5.2 Perform an in-container test for the Account Portlet

The following sections describe how to perform an in-container test for the Account Portlet with WIT.

5.2.1 In-container Test Case Naming Conventions

WIT in-container test case classes follow a specific naming convention.

Name of test case class: = Name of Portlet class + “Test”

The name of each test case starts exactly with the name of the Portlet being tested and ends with the string ‘Test’. For each Portlet method being tested, there is a pair of test methods in the test case class. The access modifier of these methods must be public, and the return type must be void. The name of these methods consists of three parts. The first part is either ‘before’ or ‘after’, and the second part is the name of methods being tested as well as all parameters in the original method, and the third part is any Java valid string to make the test methods more meaningful.

Name of test methods := public void (before | after) +
“_” + name of methods under testing +
“_” + additional string

Each *before* method will be converted into an AspectJ ‘before advice’, and each *after* method will be converted into an ‘after advice’. With AspectJ weaver, all the code in the *before* and *after* methods will be injected into the Portlet method body right before or after the original method code. The second part of a WIT test method name consists of

the method name and parameter declarations of original Portlet method. Combined with the Portlet class name, it can be used to generate an AspectJ pointcut which uniquely identifies which Portlet method is going to be wrapped.

5.2.2 Testing deployment related problems

The test scenario presented in this section allows testing for deployment related problems (see Chapter 3). The PortletSettings [JSR168 2005] object contains configuration parameters accessed by the Portlet code at runtime. These parameters are initially defined in the Portlet descriptor file called Portlet.xml. The Portal administrator uses the administrative interface to configure individual Portlet by editing the configuration parameters before deploying the application into the production environment.

For instance the accounts Portlet as shown in Figure 5.3 (line 4, 5, 6) accesses the database connection string by reading the configuration parameter from the Portlet descriptor file (refer Figure 5.4).

```
<concrete-Portlet href="#Accounts">
  <Portlet-name>Accounts</Portlet-name>
   ……
  <config-param>
    <param-name>AccountDB</param-name>
    <param-value>jdbc:db2://localhost:50000/AccountDB</param-value>
  </config-param>
</concrete-Portlet>
```

Figure 5.4: A snippet of Portlet.xml showing configuration parameters

The AccountsPortletTest code in Figure 5.5 checks for the valid database connection string in the production environment. An incorrect value read by the Portlet at runtime on the production environment will cause the AccountsPortletTest to fail. For example, DB2BOX is the name of database server used in production environment. If the Account Portlet is configured incorrectly to use some other database server, the following WIT test case will generate a failure.

```
public class AccountsPortletTest extends TestCase {
  private final String AccountDBConnStr =
                              "jdbc:db2://DB2BOX:50000/AccountDB";

  public void after_doView_testGetAcctDBConnStr
  (PortletRequest request, PortletResponse response) {
       PortletSettings PortletSettings = request.getPortletSettings();
       String dbConnStr =  PortletSettings.getAttribute("AccountDB");
      assertEquals("AccountDB Connection String is incorrect",
                dbConnStr, AccountDBConnStr);
  }
}
```

Figure 5.5: doView() – AccountsPortlet Test Case For Database Connection String

5.2.3 Automated Security Testing: Role Based Testing of Resource Access.

The test scenario presented in this section tests security privileges. We first highlight how the in-container security test case classes differ in naming convention from other test classes. A specific naming convention described below is used.

Name of security test case class: = Name of Portlet class + "SecurityTest"

Name of security test case: =”test”+ (View|Edit|Config) + “Security”

Next, we discuss an example scenario where a Portal user called David is trying to access a sensitive resource that he does not have access to.

WIT will first weave the security test case testViewSecurity () code in Figure 5.6 into the doView() of the account Portlet class. It will then login to the Portal application with the specified user name and password, and then send a request to view AccountsPortlet. If the request is successful for some reason the doView() method in AccountsPortlet will be executed – which should have been prevented by the security system. Thus, the execution of the doView method means that the security test has failed and, thus, the testViewSecurity method triggers a ‘fail’. This in turn reports a test failure to the developer.

```
public class AccountsPortletSecurityTest
               extends SecurityTestCase {
   public String getAuthenUrl() { return “http://ict5/login”; }
   public String getAuthenUser() {return “david”; }
   public String getAuthenPwd() { return “pass”; }
   public String getPortletInvokeUrl() {
        return “http://ict5/Acct”;
   }
   public String testViewSecurity() {
        fail(“the user:”+getAuthenUser()+” should not be
                      able to view the AccountPortlet”);
   }
```

Figure 5.6: doView() – AccountsPortlet Test Case For Security

5.2.4 Testing Problems Arising From the Interaction Between the Container and the Application Code

The test scenario presented in this section tests problems arising from the interaction between the container and the application code in the form of request, response objects and other application environment objects.

The AccountsPortlet depicted in Figure 5.3 displays account detail information according to the order id number submitted by the user. (line 7-14)

In Figure 5.7, developers set up the initial testing environment in the before_testGetAcctDetail method by adding an account id into the session object, and check the environment in the after_testGetAcctDetail method by comparing the outBalance with the expected number 10. If the account outbalance is not what we expected, then a failure is fired.

```
public class AccountsPortletTest extends TestCase {
  public void before_doView_testGetAcctDetail
    (PortletRequest request, PortletResponse response) {
       session.setAttribute("orderId", "123");
  }

  public void after_doView_testGetAcctDetail
     (PortletRequest request, PortletResponse response) {
     AccountDetail ad =
                   (AccountDetail) request.getAttribute("AcctDtl");
   double outBalance = ad.getOutstandingBalance();
   assertEquals("outstanding balance is incorrect", outBalance, 10);
 }
}
```

Figure 5.7: doView () Test Case – AccountsPortletTest Class

5.2.5 Using WIT To Run the Test Script

After a tester has written automated Portlet tests, he/she is able to compile, deploy and invoke all of them using a custom [Ant 2006] command that we developed. At the end of the script run, test results are displayed in the script window and the browser. The Ant based script can be integrated with the regular build process and promotes regression testing. The Accounts Portlet tests for demo usage scenario were deployed and run in the IBM Websphere Portal Server environment.

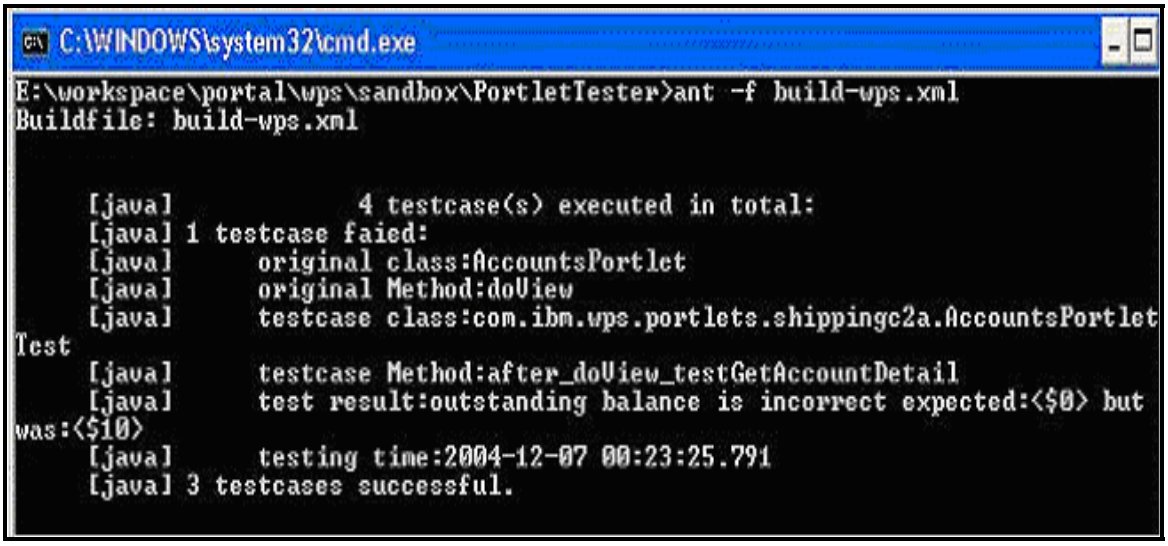

Figure 5.8: Results of Test Execution of AccountsPortletTest Cases

In addition to the text format presentation, test results also can be rendered as a browser based summary table or graph where red rows show failed tests and green rows signify tests that are successful as shown in Table 5.1.

All Test(s): Red color stands for failed tests; Green stands for successful tests

TestSuite Name	No.	Original Class	Original Method	Test Result	Testing Time
ShippingDemo	1	AccountsPortlet	doView	TotalValue is calculated correctly	2004-12-07 00:23:25.771
ShippingDemo	2	AccountsPortlet	doView	Account detail bean is not null	2004-12-07 00:23:25.771
ShippingDemo	3	AccountsPortlet	doView	account details is not null	2004-12-07 00:23:25.771
ShippingDemo	1	AccountsPortlet	doView	outstanding balance is incorrect expected:<$0> but was:<$10>	2004-12-07 00:23:25.791

Table 5.1: Test Results From Executing AccountPortlet Tests

CHAPTER SIX: QUALITATIVE ANALYSIS

The purpose of this chapter is to present a qualitative evaluation towards the value of WIT. The evaluation is accomplished by examining whether WIT has met design goals presented in Chapter 4, and comparing WIT to other related tools.

6.1 Evaluation towards design goals

The design goals of WIT are listed as:

- Accessing and controlling container managed API objects.
- Executing test code in containers
- Testing portlet methods
- Testing access to portlets
- Minimizing test execution side effects
- Script supported test

6.1.1 Accessing and controlling container managed API objects

By wrapping a method body of a Portlet with test code, WIT grants testers the ability to intercept and thus control objects like PortletRequest generated by the Portlet container. Once under control, the container objects delivering data between container and portlet are configured to set up the testing environment before they are passed into the portlet code. Test code also checks the changes on these objects caused by the execution of domain code and makes the necessary assertions.

6.1.2 Executing test code in containers

With WIT Converter and Weaver, WIT is able to translate WIT test cases into aspect code, and further inject aspect code into portlet binary code. Next, the test code is packaged into a Java .jar file and then is deployed into containers along with portlets. With WIT Invoker and Controller, test code in containers can be invoked by simulating a request for the portal page where portlets under testing are deployed. Test results from the execution of test code are collected into a centralized WIT Repository. A summary report is generated in terms of the test results saved in the repository and sent back to testers at the end of a complete in-container functional unit test.

6.1.3 Testing portlet methods

Residing inside of Portlet classes under testing, WIT test code is able to check the changes in container objects caused by the execution of portlet methods. Thus WIT is able to evaluate whether or not these portlet methods work correctly. Further, WIT test code owns the same access abilities as the other internal Portlet methods. That means private methods and fields are no more private to WIT test. The ability to test non-public methods provides testers an option to perform a finer grained test.

- Script supported test

  WIT has been fully integrated into Ant, including the setup of WIT configuration information, conversion and injection of WIT test code, deployment of test code, establishment of connection to WebSphere Portal Server, invocation to Portlets under testing, and the presentation of test results.

6.1.4 Testing access to portlets

Current design only requires WIT to support the test for authorized permissions to Portlets. Access permissions to Portlet in WebSphere Portal Server includes view, edit, config, etc, which are mapped to the doView(), doEdit(), and doConfig() methods in Portlet code. For example, a Portlet responsible for displaying an annual income report can be configured as only executive group members are allowed to browse it. WebSphere Portal Server calls doView() of the Portlet to render a report if it finds the user has been granted such permission. Otherwise, no call will be sent to doView, and hence no report shows on the page. With WIT, a tester can start an automatic process to log into a Portal application on behalf of an unauthorized user, for example David, and perform authorization-required operations such as browsing a restricted Portlet. On the server side, WIT test code running inside of doXXX() is able to obtain the authenticated user name via getRemoteUser(). The appearance of a user name like David in the doXXX() methods means a security breach exists. WIT then reports a failure in this case.

6.1.5 Minimizing test execution side effects

Since WIT test code resides in the method body of a Portlet, there are some impacts on the production code. The impacts are twofold. First, the functionalities of a Portlet may be changed because a WIT test case is able to change a Portlet's runtime environment, such as the HttpPortletRequest object that contains HTTP request parameters sent to the Portlet. Secondly, the execution performance of a Portlet is reduced because of the extra test code inside of it.

WIT has two settings to ensure that WIT test cases do not interfere with the functionalities of a Portlet. For each HTTP connection from the WIT invoker to the WPS server, WIT embeds a unique serial number string in it. WIT test code will not be activated until it receives a request with that exact string. Furthermore, at that time WIT sends out such a test request, it writes information, such as the method name, to be tested into a centralized repository. WIT test code will remain unchanged unless the information in the repository matches the name of the method where the test code resides. The second mechanism is used to specify explicitly which method is going to be tested and prevents the other methods in the same portlet from being tested. Such a combination disables WIT test code for a normal Portlet request and enables it when receiving a WIT test request. Thus, Portlets still provide the same services when a normal user is browsing pages, and meanwhile WIT test code is able to change the container environment to perform an in-container functional unit test because WPS instantiates new Portlet instance for different HTTP client request.

WIT does the above condition check in memory, an efficient way to reduce the impacts on the performance of Portlet code. The instance of HttpPortletRequest is an in-memory object generated by the container, and the repository containing information of methods to be tested is also an in-memory database. When no tests are requested, those two checks are the only places where the performance is impacted. When testing, the test code may become a bottleneck. Nevertheless, the quality of test code depends on the people, not the tool.

6.1.6 Script supported tests

WIT supports ANT scripts. This enables it to be integrated with other testing tools. With the script-supported feature, WIT can be used as part of integration testing

6.2 Comparison with related tools

WIT specializes in the test for interactions between Portlets and production containers. The purpose of performing such tests is to provide a more fine-grained way of ensuring that Portlets function properly when they are deployed and running in a production container environment. As far as we know, WIT is the only available tool at this time to support such a testing strategy. Other related tools or frameworks have limitations when dealing with this in-container testing strategy. Testing for interactions between components and the container is too coarse to report accurate data in HttpUnit, invalid in Cactus, is totally absent in JUnit. However, it is important to note that those related tools have their own strengths and can be used to test different aspects of a Portal application.

- Comparison to JUnit

  The de facto standard unit-testing tool in Java is excellent at validating code logic itself, such as algorithm checking. Its simplicity and lightweight makes it a perfect base for regression testing, which require frequent executions on a regular basis. The major limitation for this tool is its absence of ability of in container testing. For container-based domain code, such as EJB, Servlet, Portlet, etc, issues related

to the interaction between them and their containers can not be detected by this tool.

- Comparison to HttpUnit

  HttpUnit is a tool that supports functional testing for web applications. Some other similar tools exist such as jWebUnit, HtmlUnit, etc. They test web applications externally at the side of end-users by simulating browser behaviours. However, tracing the root cause of the problems they report might involve an analysis of a chain of code, such as from database access code to business logic session code to presentation code. Debugging and fixing such problems is costly. Furthermore, those tools become extremely fragile when testing WPS based applications, since the unique ID of page elements in such applications is generated dynamically.

- Comparion to Cactus

  Cactus can perform in container tests for Servlets, EJBs and JSPs, etc, but not for Portlets. Further, the in-container testing approach used by the Cactus framework is more restricted than WIT. Components, like Servlets, tested using Cactus are instantiated as normal classes in the test code versus using the real container to manage the component's lifecycle. Thus, some in-container methods and their interactions with the real container cannot be completely tested as the services provided by the real container are being in-completely used although the tests run in a real container. Cactus tests may not be able to adequately detect deployment

related errors as well as security issues, which can be, tested effectively using WIT.

- Comparison to MockObjects

  MockObjects is a tool supporting unit testing with mock object technology. By providing simulations of container objects, it isolates Portlets from the real container. This tool focuses on testing a Portlet's logic, but not its interactions with the container. Thus, test results from this kind of tools are vulnerable when Portlets run in the real container environment.

Table 6.1: A summary comparison of WIT to other tools

Features	JUnit	HttpUnit	MockObjects	Cactus	WIT
Accessing and controlling container managed API objects	No	No	Yes (By Mocking)	Yes (Partially)	Yes (Fully)
Executing test code in containers	No	No	Yes (By Mocking)	Yes	Yes
Testing portlet methods	No	No	Yes (By Mocking)	No	Yes
Testing access to portlets	No	Yes	No	No	Yes

Minimizing test execution side effect	N/A	No impact	No impact	Has minor impacts	Has minor impacts
Script supported test	Yes	Yes	Yes	Yes	Yes

CHAPTER SEVEN: CONCLUSION AND FUTUREWORK

Component-container technology is gaining more and more popularity in enterprise application development. A developer benefits from the freedom from having to implement and manage sharable services for components such as resource connections, lifecycle management, security, deployment, and threading. Hence, a developer can concentrate his/her work on delivering software features more directly involved with business functions.

Even though the advantages of this technology are obvious, it causes developers problems related to the interactions between components and containers. Components are sensitive to the container environment because they sit inside the container and rely on services the container provides to deliver functionalities. Minor changes in the container environment may result in unexpected behaviours.

The issues described above require an approach to test components when they render services in collaboration with their containers. Existing tools and approaches have different weakness in addressing these issues. Either they do not support the container environment during testing, or the container environment provided for testing is not exactly the same as the one for deployment.

A component is designed to comply with a set of contractually specified interfaces via which it communicates to their container. In other words, the interfaces define the interactions between the components and containers. In order to test such interactions, an

approach is required to intercept and control all information passed through these interfaces before it reaches component code. Such an approach provides developers a chance to set up testing scenarios by manipulating data sent to component code, and validate component-generated data before it is returned back to the container, which is what In-container Testing (ICT) is about.

A proof-of-concept tool, namely WIT, was developed to support testing for interactions between JSR168-compatible Portlets, a type of component extended from servlets, and their container built by IBM. WIT also provides support for testing authorized access of Portlets.

The design of the framework included considerations such as, how to inject test code into a component, how to minimize the impact on the production environment, and how to make the tool user-friendly. The tool makes use of the AspectJ technology, which is an aspect-oriented extension to the Java programming language created at Xerox PARC. With the AspectJ compiler, WIT test cases can be woven into Portlet byte code to implement in-container testing.

To further explore the capacity of ICT, future versions of the WIT framework could be provided with configurable settings to perform in-container testing of other container-based components such as EJBs, Servlets etc.

To highlight the strength of WIT, we provided a qualitative evaluation of WIT in Chapter 6 that shows WIT is the currently only available tool which supports in-container testing for portal applications. It would be valuable to do a more objective and finer-grained quantitative evaluation in the future. A controlled experiment may be conducted to collect data related to the usefulness and usability of WIT, such as the testing execution time in seconds.

In addition to future research work, the current implementation of the tool requires improvements as well. To trigger the execution of Portlets under testing, an HTTP connection to Portal server is established and HTTP requests follow. In the current version of WIT, two HTTP requests are generated for each API method on a Portlet under testing; one is for user authentication, and another for Portlets. In WPS, the URL of the HTTP request for Portlets is bound to API methods. In other words, the doView() method has a different URL from the doEdit() even for the same Portlet. Such requests are very time-consuming. To reduce the number of HTTP requests, performance tuning is required. Apparently, invoking all rather than one API methods with one HTTP request will accelerate the testing process. The authentication requests can be reduced by simultaneously testing multiple Portlets on the same page after a user is authenticated. The current version has to invoke a user authentication process for every Portlet no matter if it is deployed on the same page as some other Portlet.

The usability of the current implementation should be improved as well. To perform in-container testing, each method of a Portlet is configured with parameters for instance the

URL link to a page into which it is deployed. Currently, these parameters are manually put together by testers in an XML file to configure the testing environment. It would be beneficial to provide an eclipse plug-in to support an automatic configuration and invocation of in container testing.

CHAPTER EIGHT: REFERENCES

[Ant 2006] Apache ANT, http://ant.apache.org/, accessed February 7, 2006

[AspectJ 2006] AspectJ, http://www.eclipse.org/aspectj/, accessed May 10, 2006

[Bass 2000] Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, Kurt Wallnau, Volume I: Market Assessment of Component-Based Software Engineering, Software Engineering Institute, CMU/SEI-2001-TN-007, May 2000, p21-24, http://www.sei.cmu.edu/publications/documents/01.reports/01tn007.html, accessed June 21, 2006.

[Bass 2006] Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, Kurt Wallnau, "Volume I: Market Assessment of Component-Based Software Engineering", pp. 32, http://www.sei.cmu.edu/publications/documents/01.reports/01tn007/01tn007.html#chap01, accessed January 17, 2006

[Bosch 2000] Bosch, J.: Design and use of software architectures: Adopting and evolving a product-line approach. Addison Wesley. 2000

[Cactus 2005] Cactus Apache Jakarta Project, http://jakarta.apache.org/cactus/, accessed January 29,2005

[Clements 2001] Clements, P. and Northrop, L. (2001). Software Product Lines: Practices and Patterns, New York, Addison-Wesley.

[COM 2006] COM: Component Object Model Technologies, http://www.microsoft.com/com/default.mspx, accessed January 17, 2006

[EasyMock 2006] EasyMock, http://www.easymock.org/, accessed March 25, 2006

[EJB 2006] Enterprise JavaBeans Technology, http://java.sun.com/products/ejb/, accessed January 17, 2006

[Holser 2006] Paul Holser, Limitations of reflective method lookup, http://www.adtmag.com/java/article.aspx?id=4276, accessed May 20, 2006

[HSQLDB 2006] HSQLDB, http://hsqldb.org/, accessed March 21, 2006

[HttpUnit 2006] HttpUnit, http://www.httpunit.org/, accessed March 20, 2006

[HTML 2006] HyperText Markup Language, http://www.w3.org/MarkUp/, accessed May 10, 2006

[Hunt 2001] Andy Hunt and Dave Thomas, Pragmatic Unit Testing, Chapter 6, p65, ISBN: 0-9745140-1-2

[IBM 2006] Using JSR 168 with WebSphere Portal, http://publib.boulder.ibm.com/pvc/wp/5021/ent/en/standards/jsr168.html, accessed March 25, 2006

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

[J2EE 2006] Java Platform, Enterprise Edition (Java EE), http://java.sun.com/javaee/technologies/, accessed January 17, 2006

[J2SE 2006] JavaTM 2 Platform, Standard Edition, v 1.3.1 API Specification, http://java.sun.com/j2se/1.3/docs/api/index.html, accessed May 20, 2006

[JFCUnit 2003] JFCUnit, http://jfcunit.sourceforge.net/, accessed March 20, 2006

[JSR168 2005] JSR-000168 Portlet Specification, http://www.jcp.org/aboutJava/communityprocess/review/jsr168/, accessed February 11, 2005

[JSR154 2005] Java Servlet Specification 2.4, http://www.jcp.org/aboutJava/communityprocess/final/jsr154/index.html, accessed February 11, 2005

[JSR245 2005] JavaServer Pages 2.1, http://www.jcp.org/aboutJava/communityprocess/edr/jsr245/index.html, accessed February 11, 2005

[JUnit 2005] JUnit, http://www.junit.org/index.htm, accessed June 20, 2005

[JunitEE 2006] JunitEE, http://www.junitee.org/, accessed March 20, 2006

[Kent2002] Kent Beck, Test Driven Development: By Example, Addison-Wesley Professional, 2002

[Kim 2005] Haeng-Kon Kim; Lee, R.Y.; Hae-Sool Yang; Software Engineering Research, Management and Applications, 2005. Third ACIS International Conference on, pp. 375 – 382, 11-13 Aug. 2005

[LazyValidatorForm 2006] LazyValidatorForm class, http://struts.apache.org/struts-doc-1.2.7/api/index.html, accessed May 10, 2006

[Maurer 2000] Peter M. Maurer, "Components: What If They Gave a Revolution and Nobody Came?," *Computer*, vol. 33, no. 6, pp. 28-34, Jun., 2000.

[MockMaker 2006] MockMaker, http://www.mockmaker.org/, accessed March 25, 2006

[MockRunner 2006] MockRunner, http://mockrunner.sourceforge.net/index.html, accessed March 25, 2006

[MockEJB 2006] MockEJB, http://www.mockejb.org/, accessed March 25, 2006

[MockObjects 2006] MockObjects, http://www.connextra.com/aboutUs/mockobjects.pdf, accessed March 20, 2006

[MyYahoo 2006] My Yahoo, http://my.yahoo.com/, accessed June 24, 2006

[PatternTesting 2005] PatternTesting, http://patterntesting.sourceforge.net/index.html, accessed September 3, 2005

[Pfleeger 1998] Software Engineering: Theory and Practice. Shari Lawrence Pfleeger. ISBN 0-13-624842-X

[PortletUnit 2006] PortletUnit, http://Portletunit.sourceforge.net/, accessed March 25, 2006

[Reflection 2006] The Reflection API, http://java.sun.com/docs/books/tutorial/reflect/index.html, accessed March 21, 2006

[Saha 1999] Avi Saha, Application Framework for e-business: Portals, http://www-128.ibm.com/developerworks/library/wa-portals/index.html, accessed June 24, 2006

[Sridhar 2006] Nigamanth Sridhar, Jason O. Hallstrom, Generating Configurable Containers for ComponentBased Software, http://www.csse.monash.edu.au/~hws/cgi-bin/CBSE6/Proceedings/papersfinal/p23.pdf, accssed July 26, 2006

[StrutsTestCase 2006] StrutsTestCase for Junit, http://strutstestcase.sourceforge.net/, accessed March 20, 2006

[Struts 2006] Struts Framework, http://struts.apache.org/, accessed March 20, 2006

[Szyperski 1998] C. Szyperski, Component Software, Beyond Object-Oriented Programming, Addison-Wesley, 1998.

[Tushar 2002] Tushar K. Hazra, Building Enterprise Portals: Principles to Practice. Proceedings of the 24rd International Conference on Software Engineering, 2002 Page(s): 623 – 633

[UnitTest 2006] Unit Test, http://en.wikipedia.org/wiki/Unit_test, accessed July 26, 2006

[WAS 2006] WebSphere Application Server, http://www-306.ibm.com/software/webservers/appserv/was/, accessed June 20, 2006

[Watir 2006] Watir, http://wtr.rubyforge.org/, accessed March 25, 2006

[Web 2006] The World Wide Web Consortium (W3C), http://www.w3.org/, accessed June 24, 2006

[Wege 2002] Christian Wege, DaimerChrysler. Portal Server Technology, IEEE Internet Computing 2002

[WPS2005] Portal Introduction-IBM, http://www-106.ibm.com/developerworks/ibm/library/i-Portletintro, accessed February 7, 2005).