About IBM Software Services for WebSphere

IBM Software Services for WebSphere is a team of highly skilled consultants with broad architectural knowledge, deep technical skills, best practices expertise, and close ties with the IBM research and development labs. As an integral part of all software sales, we provide worldwide support for WebSphere products through a set of services that make it easy to design, build, test, and deploy applications for e-business. Our purpose is your success. To learn more about us, please visit our Internet site at:


About IBM Software Services for WebSphere Best Practice Guides:

Welcome to Version 0.9! This is a collection of useful information from our most talented WebSphere consultants. These documents don't represent the complete art on any of these topics. (Goodness knows, we could write a book on any of these topics.) Rather, this is the start of sharing our best practices with you, to make you more successful. We will continue to update these documents, add more topics, expand these topics, and publish them to the WebSphere Developer Domain.

http://www7b.software.ibm.com/wsdd/
# Table of Contents

Introduction ......................................................................................................................... 5  
Intended audience ............................................................................................................... 5  
Document organization ..................................................................................................... 6  
Where do I start? ................................................................................................................. 6  
Overview of the Rational Unified Process ........................................................................ 7  
The test plan ....................................................................................................................... 8  
Construction phase testing .............................................................................................. 9  
  Construction phase test plan .......................................................................................... 10  
  Use case review ............................................................................................................. 11  
  Design model review ..................................................................................................... 11  
  Code testing .................................................................................................................. 11  
  Performance testing ....................................................................................................... 13  
  User interface testing ................................................................................................... 15  
  Code review .................................................................................................................. 16  
  Regression testing ........................................................................................................ 16  
  Defect tracking ............................................................................................................ 17  
Transition phase testing .................................................................................................. 18  
  Transition phase test plan ........................................................................................... 19  
  Function testing .......................................................................................................... 19  
  Stress testing ............................................................................................................... 19  
  Installation testing ....................................................................................................... 21  
  Operations testing ....................................................................................................... 21  
  Alpha/beta/pilot testing ............................................................................................... 23  
  User acceptance testing .............................................................................................. 23  
  Regression testing ....................................................................................................... 23  
  Defect tracking ............................................................................................................ 24  
Application Testing Triage ............................................................................................... 25  
Summary ............................................................................................................................ 26  
Appendix A: Glossary ........................................................................................................ 27  
Appendix B: Tools ............................................................................................................ 28  
  Design (use case analysis) ............................................................................................ 28  
  Test case management ................................................................................................. 28  
  Code testing tools ........................................................................................................ 28  
  Automated Web testing tools ..................................................................................... 28  
  Version control ............................................................................................................ 29  
  Defect tracking ............................................................................................................ 29  
Appendix C: Topics for further consideration ................................................................. 30  
  Case Study ................................................................................................................... 30  
  Application Profiling .................................................................................................. 30  
Bibliography ....................................................................................................................... 31  
References ......................................................................................................................... 32
Introduction

The Java 2 Enterprise Edition (J2EE) specification provides a well-documented programming model for the development of distributed, component-based software systems using Java. Though the specification is quite straightforward, there are still some inherent complexities in large-scale distributed systems development.

To mitigate these complexities, the J2EE programming model took a divide-and-conquer approach, which led to the design of the J2EE component architecture (see Figure 1). One of the goals of the J2EE architectural componentization was to more clearly delineate the boundaries between the various development roles defined for the J2EE application programming model (for example, HTML document designers, document programmers, and enterprise bean developers).

A by-product of this desire to separate concerns for the various development roles is an increase in the complexity of component integration. This means there are now more “moving parts” in the system that must connect and interoperate correctly. But how do you verify this correctness? The answer: testing. Testing applications based upon the J2EE component architecture is a difficult task that requires a high level of discipline on the part of all parties involved in the development process. This document provides a roadmap to help those responsible for testing WebSphere® J2EE applications.

Intended audience

Every system stakeholder should read this document, including development, operations, and management. This document assumes the reader is familiar with the Rational Unified Process (RUP) software development lifecycle process [Jacobsen, Booch, Rumbaugh] and the Unified Modeling Language (UML).

Figure 1: J2EE logical component architecture
Document organization

The document begins with a brief introduction to RUP in general, and use cases in particular, followed by a discussion of the importance of the test plan document. These two concepts are discussed first because they fulfill the following testing prerequisites:

- You must understand the system you are preparing to test. By definition, testing is the verification of function. If you don’t know the function, you can’t test it. Within RUP, the use case represents function.

- The test you are preparing must be repeatable so that results can be compared. A test plan ensures that the tests can be repeated.

The rest of the document discusses the various testing activities associated with the two final phases of RUP: the construction and transition phases.

Where do I start?

The answer depends upon your situation. This document focuses on the construction and transition phases of RUP. If your application is still under development and has not yet been released for user testing, begin with the testing workflow activities associated with the construction phase (see the “Construction phase testing” section).

If you are in the process of releasing your application to users, begin with the activities in the transition phase (see the “Transition phase testing” section). Be sure to go back to the construction phase testing for your next iteration.

If you are going to production within the week and you feel your testing has been inadequate, view the “Application testing triage” section.
Overview of the Rational Unified Process

The Rational Unified Process (RUP) is a software development process for developing component-based systems that are connected by well-defined interfaces [Jacobsen, Booch, Rumbaugh]. RUP has three major characteristics:

- Use-case driven
- Architecture-centric
- Iterative and incremental.

RUP uses the Unified Modeling Language (UML) for expressing the software system models, of which use cases are a vital artifact. A use case is defined as some functionality in the system that provides value to some user, whether that user is a human or another system.

RUP has four distinct phases: inception, elaboration, construction, and transition (see Figure 2). Each phase represents the span of time between two major milestones of a development process.

Within each phase, five workflows have been defined: requirements, analysis, design, implementation, and test. Workflows are defined as activities that realize a part of a business use case. Thus, the binding concept used within the unified process is the use case. All other artifacts and activities within the unified process depend upon the business use case and its associated actor(s). This document focuses on the construction and transition phases, the phases in which most testing workflow activity occurs.

Figure 2: Unified Process phases and workflows

Given a universally accepted task list, the RUP focuses on the analysis of user stories, or how users would like to see the output of their system. The RUP is a software development process for component-based systems that are connected by well-defined interfaces.
The test plan

A test plan is a written document that describes your organization's testing strategies, resources and schedule. It is not a conglomerate of JUnit classes with a ReadMe file (though this may comprise an artifact of your test procedures). No matter what the level of maturity of your testing model, a test plan is a must.

The creation of this document depends heavily on your use cases. Remember, testing requires verification of function, and in the Unified Process, function is expressed through use cases. You first create your test cases from your use cases, and then apply user priorities to set precedence for the test cases. Below is a series of suggested steps for creating test cases [McGregor, Sykes]:

1. Define a complete set of actors.
2. Define a complete set of use cases, including associations with actors.
3. Construct the use profile for each actor.
4. Compute the frequency attribute for each use case from the actor use profiles.
5. Combine the frequency and criticality ratings for a single use case into a "test priority" value.
6. Allocate test cases based upon the test priority of the use case.

Upon completion of this task, you will have a list of prioritized test cases. You now have the key information needed to guide the development of the rest of the test plan (which must include construction phase and transition phase roll-ups), as well as all artifacts you will need to support it (for example, JUnit scripts, Web load scripts, and review documents).
Construction phase testing

The construction phase is the third phase of the Unified Process software life cycle. During this phase, the software is brought from an executable architectural baseline to the point where it is ready to be transitioned to the user community [Jacobsen, Booch, Rumbaugh].

Figure 3 illustrates the testing activities associated with the construction phase. Each of these testing activities is discussed in this section, with specific hints based upon the J2EE architecture and WebSphere®. Also note that in the testing workflow, two overlapping activities are constantly occurring: regression testing and defect tracking. Regression testing is the process of verifying that existing functionality in prior builds still works when new functionality is added in new releases. Defect tracking is the process of recording problems discovered during testing, or during a review. The testing activities include reviews of documents, such as the use cases and design models. In RUP, models are deliverables, just as software and executables are.

![Figure 3: Construction phase testing](image)

The Unified Process stresses the concept of iteration and parallel activity across workflows. The testing workflow occurs in parallel within a phase with the other workflows (see Figure 2), including implementation; however, it is beneficial to perform some testing activities prior to others to prevent re-work. Many development projects assume testing is the last activity in a development iteration, but if development does not perform the use-case review prior to implementation, you may find that you are building a system that does not meet your specifications. Within the construction phase’s testing workflow, there is a desired order of activities. These activities are illustrated in the testing workflow shown in Figure 4, which is depicted as a UML activity diagram.
Figure 4: Testing workflow activity diagram

Each of the states in the activity diagram are, of course, optional, but so is testing in general—if you are willing to severely jeopardize the chances of your project’s success. The activity diagram illustrates those activities that must come first, such as use case review and design model review. The diagram also shows which activities can occur in parallel, such as performance testing, user interface testing, and code review. Notice that defect tracking and regression testing occur in parallel throughout the testing workflow.

Note that the activity diagram’s initial state depends on the existence of the master test plan document, as does the development of the construction-phase-specific test plan (which is a component of the master test plan).

Construction phase test plan

You must develop a test plan for the construction phase. This test plan is a roll-up to the master test plan; it simply details the process of testing during the construction phase, which allows for a repeatable process for verifying system function. The document should be considered a deliverable of the construction phase, with as much importance as program code, because the code only has value if it has been verified for correctness.

Figure 3 details some of the testing workflow activities associated with the construction
phase. Your test plan should include a description of the activities you choose to incorporate into your test plans.

**Use case review**

The use case review should occur early in the construction phase, prior to the implementation activities associated with a particular use case. The test case for this process is as simple as stating that a business analyst and a designer will review the use case for correctness, updating the test plan to show that the use case is ready for the implementation workflow.

**Note:** It is a good idea to include production support personnel in the use case review process. This fosters open communication between operations and development, which is invaluable during the defect tracking activity in the transition phase.

**Design model review**

The design model review involves the designers and the system architect(s). Remember, the Unified Process is architecture-centric; thus, this review provides a vital point to verify the correspondence between the design and the architecture.

**Code testing**

This is the activity traditionally associated with software testing and includes traditional testing methods such as black box, white box, gray box, coverage, and boundary. Performing code testing on J2EE applications can be difficult—the J2EE programming model requires separation of concerns, which means a large number of individuals must be involved in the code testing. How do you organize your testing activities to ensure proper coverage and minimize time? The figure below is a UML activity diagram depicting the activities associated with J2EE application development, along with the J2EE programming roles responsible for each activity (simply look at those activities directly under the roles). Within the diagram, various test points have been highlighted to identify points in the development process where complete testing scripts should be created to verify the function of the deliverables, as well as the roles responsible for creating the scripts.

**Note:** For more information about the WebSphere development environment and its support of the J2EE programming model, see “Best Practice: WebSphere Application Development Infrastructure” by Matthew Oberlin.
Figure 5: J2EE functional testing points

Tasks

This section discusses the tasks associated with each of the testing points illustrated in the J2EE functional testing points diagram (Figure 5). The task discussions are ordered according to any dependencies that exist among them, with the exception of the HTML testing task, which has no dependencies.

Backend resource testing depends on the backend resource being used. However, you should use tools such as JUnit and Java to create automated tests for these resources. Because of the desire to automate the testing through Java-based tools, many times the enterprise bean developer works with the resource developer to develop the initial test scripts. In the case of RDBMS resource, tests should exercise any stored procedures and/or triggers used.

Enterprise bean testing is the responsibility of the bean developer, and it may include both entity bean and session bean testing. If a session bean depends upon an entity bean, then of course the entity bean test must succeed prior to running the session bean test. The enterprise bean level tests must incorporate J2EE descriptor concepts such as security. This means that if roles are defined by the enterprise bean developer, then exception tests should be run to verify expected behavior. This highlights another important point in testing with J2EE and WebSphere.
Note: Make sure security is configured the same way in all testing activities as it will be in QA/production. Do not disable security in the construction phase with the idea of turning it on in the transition phase.

JSP testing is more complicated than HTML testing and may require both an HTML document designer and a document programmer if tag libraries and/or scriptlets are needed. JSP testing first requires that the document programmer verify the function of tags and or scriptlets; the HTML designer then verifies the overall look and feel of the generated HTML page. The first iteration of JSP testing is a manual process, but subsequent iterations can be automated using a Web-testing tool.

HTML testing simply entails verifying the content and requirements (for example, corporate standards for look and feel) of all HTML pages. These activities are manual for the first testing iteration, but can be automated in subsequent iterations using the Web-testing tool.

Performance testing

Performance testing verifies that the system will perform properly under various load conditions. These varying load conditions include base performance testing, load testing, and stress testing. Each performance-related testing activity is shown in Figure 4 and discussed in the following sections.

A good Web-testing tool is a must for the performance testing activity; “Appendix B: Tools” provides a list of recommended tools. The tool should allow for the creation (usually recording) of scripts, which should provide adequate application coverage (meaning all components—such as enterprise beans and JSP pages—are used). It is also important that the Web-testing tool support multiple virtual users, which will exercise the session engine to provide accurate results for session table access times and CPU utilization on the session database (if using persistent sessions). Most tools also provide the ability to store the resulting metrics for comparative purposes during regression testing.

It is very important to monitor several different components of the WebSphere system during performance testing:

- Memory usage on the WebSphere Application Server node(s).
  WebSphere Resource Analyzer is a valuable tool for monitoring this information.
- Disk space usage, such as overly verbose logs.
- Application error logs (silent or masked exceptions that the application mistakenly does not propagate).
- CPU usage on the WebSphere Application Server node(s).
- CPU utilization on the session database node (if persistent sessions are used).
- Backend resources used, such as RDBMS or MQSeries.
NOTE: IBM offers a training course on performance testing: IBM WebSphere Performance Tools & Methodology Workshop (SW242).

Base performance testing
Base performance testing measures response times and resource utilization as a single virtual user goes through the test case under ideal system conditions. This test measures the expected ideal response times of the system. The metrics gathered are used to verify that the system meets the minimal performance requirements; if it doesn’t, a defect must be recorded.

Load testing
Load testing is perhaps the single-most important testing workflow activity after functional testing (keep in mind that it does matter if your system does the wrong thing really fast). The goal is to simulate expected system usage to measure response times and system resource usage. Again, a good Web-testing tool is a must.

Note: Load testing is perhaps the single most important testing workflow activity after functional testing.

During testing, the load applied to the system usually starts off fairly light, and then is incrementally increased. As the load increases, system resource usage increases. The metrics from this test can be used for capacity planning (for example, determining the number of servers required, as well as CPU speed, memory, and disk space requirements).

During load testing, WebSphere session management must be configured in the same way it will be configured in production. If the configuration differs between load testing and production, performance metrics will be inaccurate. In addition, latent bugs related to servlet session management will remain hidden until the transition phase testing, at which point the fix is much more expensive.

Note: Do not, under any circumstances, turn off persistent sessions in load testing if they will be used in production.

Stress testing
The goal of stress testing is to push the system to its limits and observe the resulting system behavior. The limit depends on the resources available to your system. Stress testing helps answer questions about performance and behavior (Is database latency the cause of the performance bottleneck, or is it the amount of memory on the server? Does the system degrade gracefully, or does it blow up?), and determines the limits of the system. As a result, capacity planning is easier and more accurate.

Note: Accurate stress testing requires the successful completion of the load testing activity.

The Web-testing tool scripts used during the previous base performance and load-testing activities can be reused for the stress tests.
User interface testing

User interface (UI) testing for J2EE applications most often involves testing through the browser component (see Figure 1). The browser makes requests upon the Web container, which in turn may use other components of the J2EE server to perform tasks and return content. Several aspects of Web application user interface testing are listed below:

- UI standards testing
- Browser compatibility testing (Netscape, Internet Explorer, etc.)
- Exception/error management testing (critical, and very difficult).

UI standards testing

UI standards testing simply entails a review of the HTML and/or JSP pages to ensure their adherence to defined standards. In order to conduct UI standards testing, your organization must have a set of defined standards. If your organization has not defined standards, then you should make it a priority to do so for several reasons. The first and most important reason is that users will appreciate and benefit from UI standards across applications. The second reason is to reduce the difficulty associated with Web application maintenance. Web application UI development is already a fairly complex process (even with frameworks such as STRUTS); there is no need to increase complexity with UI variance.

Browser compatibility testing

Browser compatibility testing is a difficult and timely task, particularly if standards were not predefined to mitigate many of the issues. These issues include HTML-specific issues (such as layers and font management), as well as issues with differences in JavaScript interpretation. Many organizations require a specific browser version for a given application to greatly simplify and reduce the time required for testing. This solution may be feasible for intranet applications, but Internet applications may require more testing; timelines need to be adjusted accordingly.

Exception/error management testing

Exception/Error management in Web applications is a difficult task, particularly where issues associated with form data validation are concerned. Because of the stateless nature of the HTTP protocol, it is not trivial to perform such tasks as refreshing a posted form that indicates which fields are invalid and why. There are extensions built upon the STRUTS framework to aid in this process, but each of the scenarios must first be identified, and then test cases must be created. Test cases will be created based upon the exception flows within the system use cases. Each exception flow should be tested
through the browser component for proper handling. The amount of time required to perform proper UI exception management testing should not be underestimated.

UI testing activity for J2EE applications occurs last in the testing workflow (see figure 5) because the implementation of the Web application hosted in the Web container depends on all other code artifacts developed for the application (for example, enterprise beans and data sources). An error, as seen from the browser component, could be caused by a simple JSP page, servlet, or enterprise bean, or it could stem from within a database or other remote data source. In the case of WebSphere, it could also be an improperly configured plug-in, or perhaps some other issue between the Web server and the Web container. The reality is that no useable UI test results will be obtained until the code testing activity has been completed.

**Code review**

Most developers do not view code reviews as testing, but there are several reasons that it should be included as part of your test plan:

- It is a tremendous tool for discovering latent errors in software. It is amazing how a person who stares at the same set of code hundreds of times will miss what someone else who quickly scans the code would identify as a blatant bug. This is not a shortcoming of the developer; it is merely a by-product of the human condition. A different set of eyes can prove invaluable for identifying coding errors.
- It provides a wonderful opportunity to perform code *refactoring*. Refactoring is the process of incrementally improving software through an iterative process of design and implementation. Each piece of code should be revisited to see if design decisions are valid, or if improvements can be made. The best suggestion at this point is that simple is better.

The following list depicts the most troublesome issues associated with J2EE development.

- Session management using servlet API.
- Resource management (allocating and releasing resources such as JNDI or JDBC Connections).
- Exception management in enterprise beans (consider transactional semantics when declaring and throwing exceptions).
- Lack of trace (almost any bug can be identified if the system has the proper level of trace code).

**Regression testing**

Regression testing is the process of verifying that existing functionality in prior builds still works when new functionality is added in new releases. Construction phase regression testing can be automated with the help of tools such as ANT, JUnit, and Web
testing tools (see “Appendix B: Tools” for more information). For this reason, testing should be run as frequently possible—nightly is preferable. The automated regression tests should incorporate the code testing and stress testing activities, the results of which should be stored and used for comparative purposes and as input for defect tracking.

**Defect tracking**

Defect tracking is the process of recording problems discovered during testing or during a review. The development teams, who are also responsible for creating the test scripts, are responsible for construction phase defect tracking. For this reason, defect tracking in the construction phase is much more straightforward than defect tracking in the transition phase, where end users are responsible for creating defect reports that the development staff must interpret.
Transition phase testing

The transition phase is the fourth phase of the Unified Process software life cycle. During this phase, the software is turned over to the user community [Jacobsen, Booch, Rumbaugh]. Figure 6 illustrates the testing activities performed during the transition phase. Each of these testing activities is discussed in this section, with specific hints based upon the J2EE architecture and WebSphere.

As in the construction phase, the transition phase testing workflow has the constant regression testing and defect tracking activities. As stated before, the Unified Process stresses the concept of iteration and parallel activity across workflows. The testing workflow occurs in parallel within a phase with the other workflows (see Figure 7), including implementation; however, it is beneficial to perform some testing activities prior to others to prevent re-work.

As Figure 7 below illustrates, operations testing should follow installation testing—the preferred mechanism for operations to receive new releases is through an installable package that is delivered by development. This is an often-neglected activity that inevitably is the cause of much of the inner turmoil among development and operations staff associated with WebSphere installations. In the optimal process, the operations team receives an installable package from development, along with an installation procedure. Following this guideline will reduce the chances of configuration issues (such as one node in a cluster with a different release than another node—a situation that can lead to intermittent failures due to plug-in load balancing algorithms) that are very difficult to troubleshoot.
Transition phase test plan

You must develop a test plan for the transition phase. This plan is simply a roll-up to the master test plan; it provides details of transition phase testing and serves as record that allows repeatable tests to verify system function. The document should be considered a deliverable of the transition phase. Figure 7 details some of the testing workflow activities associated with the transition phase. Your test plan should include a description of the activities you choose to incorporate into your test plans.

Function testing

Function testing requires the application developers to verify that the application meets user requirements specified during analysis. In the case of J2EE applications, this test is usually performed through the browser component, whereby the developer walks through each use case developed during the construction phase to verify function. This proves to be an opportune time to record the browser interaction through the use of a load-testing tool, which will be used during the subsequent stress testing activity.

Stress testing

Stress testing J2EE applications in the transition phase should be done through the browser component, and should include all of the components that will exist in the production deployment, including network components such as firewalls and load balancers. Inevitably, these components change the behavior of the system, giving rise to problems such as the following:
Session timeout mismatches between load balancers and WebSphere sessions.
Network issues related to ports not being opened through the firewall that the WebSphere plug-in uses to talk to the WebSphere application servers.

A Web-based load-testing tool, such as Mercury LoadRunner, should be used to provide the stress. You may be able to reuse the scripts created for function testing.

**Note:** It cannot be emphasized enough that transition phase testing must occur in a clean test environment that is as close to the production environment as possible.

Here are some tips for your WebSphere testing environment:

- Use a physical infrastructure that is as similar as possible to the infrastructure that will be used in production. This includes all hardware (such as load balancers and firewalls).
- Ensure your testing environment is on an isolated network. Web application testing is tough enough—don’t allow for the possibility to introduce network issues as well.
- Ensure that backend resources are not shared. For example, make sure the WebSphere Application Server session database is not being used for another application, unless that is how it is to be used in production.
- If possible, have the same number of applications hosted on a node as you plan to have in production. Don’t test with a single application on a node if you plan to have ten other applications on the node in a production environment.
- Use the production LDAP server if at all possible. Many users use test LDAP servers with much fewer entities, which can lead to false response times when security is turned on in WebSphere.

It is very important to monitor several different components of the WebSphere system during testing:

- Memory usage on the WebSphere Application Server node(s). WebSphere Resource Analyzer is a valuable tool for monitoring this information.
- Disk space usage, such as overly verbose logs.
- Application error logs (silent or masked exceptions that application mistakenly does not propagate).
- CPU usage on the WebSphere Application Server node(s).
- CPU utilization on the session database node (if persistent sessions are used).
- Backend resources used, such as RDBMS or MQSeries.
Installation testing

Installation testing is the process of verifying that your application can be installed properly. In general, development should provide an installation package with supporting documentation to operations, which in turn installs the application using the supplied documentation. The installation test verifies this process.

This test is often ignored, and can result in obscure errors in production. For example, if two different nodes in a WebSphere domain have different releases, the resulting errors would be difficult and time consuming to troubleshoot. It is also a good practice to make the installation as automated as possible to reduce the risk of human error.

If the operations team cannot successfully install the application by using the package and documentation provided by development, then the installation test fails and the defect should be registered.

Note: Do not make the mistake of allowing development to troubleshoot and remedy production (or even QA) installations.

Operations testing

Operations testing verifies that the requirements of the operations personnel who are responsible for supporting/operating the applications in various environments are met. This includes such tasks as installation (see the “Installation testing” section above) and deployment model testing, as well as application supportability testing. The supportability testing includes operation’s ability to detect errors and collect diagnostic information for troubleshooting.

The Unified Process prescribes the use of a deployment model that “defines the physical system architecture in terms of connected nodes” [Jacobsen, Booch, Rumbaugh]. This model is comprised of deployment diagrams, which illustrate a view of the runtime configuration of processing nodes and the components that run on those nodes. Referring back to Figure 1, the logical architecture of J2EE is fairly complex. However, this does not convey the true deployment model issues, particularly in the case of WebSphere. For example, a standard, single-domain, dual-node topology is illustrated in Figure 8.
Figure 8: A single-domain, dual-node WebSphere topology

This is a very common topology, and one that is even more complex than the logical J2EE component architecture. No less than four nodes are involved, with approximately 13 processes running (and possibly more, based upon platform and chosen runtime components). This diagram also excludes the possible firewalls between the HTTP servers and the application servers, as well as any data source nodes required by installed applications. After viewing this diagram, it becomes clear that there are a significant number of break points in the system, excluding the application software itself.

Note: More information on the WebSphere operational environment can be found in the WebSphere Application Server Operations white paper, by Peter Van Sickel.

Where do you begin your deployment model testing workflow? I suggest you take the same approach as with all other testing workflow activities: Ask yourself, "What am I most concerned about?" The usual response is fail over, or workload management. These are important criteria, but when developing your test-case priority list, always think of the function that is most critical (that is, if this function isn't working, nothing else matters).

Some of the suggested test cases are listed below. Each should result in no perceivable problem from the browser component (Web test tool), based upon the WebSphere topology shown in Figure 8.

- Node A or B becomes unavailable (unplug the network adapter).
- The application server stops on a node (Kill the application unkindly).
- The Web server stops (kill a Web server).

**Alpha/beta/pilot testing**

These are the standard end-user testing phases. Not all requirements will be captured during the elaboration phase and these tests are critical for filling in gaps in the analysis model. The critical issue with this end-user testing activity is the tester’s ability to report the defect in a meaningful manner so that development can reproduce the failure. It is for this reason that the test plan must have well-documented test cases, particularly for the Web browser test cases.

**User acceptance testing**

User acceptance testing verifies that an application meets the end users’ needs. This usually entails once again verifying the use cases through the browser component. The user acceptance test should occur on a system that is currently realizing predicted system load, to verify that response times meet the end users’ requirements. The load scripts used for stress testing can be reused for applying load during user acceptance testing.

Particular care should be taken to ensure cache issues do not occur during the test. It is generally a good practice to delete the cache completely before beginning the browser test cases.

*Note*: User acceptance testing must be done while the application is under predicted system load.

**Regression testing**

Regression testing is the process of verifying that existing functionality in prior builds still works when new functionality is added in new releases. Transition phase regression testing is often a difficult and time-consuming process. As opposed to construction phase regression testing, many of the transition phase testing activities cannot be automated and require an extensive amount of time (for example, user acceptance testing). It is for this reason that regression testing in the transition phase is performed only when a new release comes from the construction phase iteration. Several different events can trigger a release from the construction phase:

- Application change (bug fixes or enhancements).
- Platform change (WebSphere version, database, or operating system).

Neither of these events should be taken lightly. Because regression testing in the transition phase is so time consuming, every effort must be made to improve and hone this process.
Defect tracking

Defect tracking is the process of recording problems discovered during testing or a review. Transition phase defect tracking is done by both operations staff and end users. It is more difficult and time consuming than construction phase defect tracking because neither group is intimately associated with the application internals. The biggest issue becomes the ability to accurately report the defect back to development, and development's ability to reproduce and track down the issue from the provided defect information. The time required for this process must not be underestimated.
Application Testing Triage

This section is targeted at individuals who are in a WebSphere application development environment where the testing process is less than adequate, the application is in either QA or production, and unexplainable errors are occurring. Several different scenarios lead to the necessity for a testing triage, the most common being a looming deadline that provides inadequate time for proper testing (I have yet to encounter a Web application development project not suffering from this trait). What does the tester do?

The best solution I have found to this problem is load testing the application in an environment that is as close as possible to the anticipated production environment. Once functional correctness has been verified, load testing provides the best opportunity to discover latent bugs in the system. These bugs range from inadequate disk space for logs (or inordinately large amounts of trace) to session persistence problems related to improper session-related coding techniques (a very common source of problems). Load testing in general will quickly flush these problems out. However, the difficulty lies in the fact that the symptoms associated with these bugs are sometimes quite difficult to diagnose. It usually requires the aid of development to help troubleshoot these error conditions.

Load-testing tools such as Mercury's LoadRunner prove invaluable for application testing triage. The ability to simulate multiple virtual users with varying amounts of "think time" allows for the most accurate reproduction of actual system load.

**Note:** Proficiency with the chosen load tool is a must, as it is used throughout the testing workflow.
Summary

Testing in general is difficult; J2EE application testing with WebSphere is an even more complicated task, involving many stakeholders within the organization. This document has provided a roadmap for planning your own testing process by identifying the sequencing and dependencies of the varying testing activities, as well as the roles of those involved. It may not be feasible for your organization to implement each of the activities outlined due to time or resource constraints, but remember: The price of not testing is usually much higher.
Appendix A: Glossary

activity: A standard unit of work performed by a worker in a workflow.

activity diagram: A UML diagram that describes the sequencing of activities, with support for both conditional and parallel behavior.

construction phase: The third phase of the Unified Process software life cycle. During this phase, the software is brought from an executable architectural baseline to the point where it is ready to be transitioned to the user community.

defect tracking: The process of recording problems discovered during testing, or during a review.

deployment model: A UML model that defines the physical system architecture in terms of connected nodes and their associated processes.

document designer: A J2EE programming role responsible for design of HTML and JSP pages (the “look and feel”). Does not write scriptlets or tag libraries (see document programmer).

document programmer: A J2EE programming role responsible for developing scriptlet code and/or tag libraries for JSP pages.

enterprise bean developer: A J2EE programming role responsible for development of enterprise beans.

load test: A test used to simulate expected system usage to measure response times and system resource utilization.

Rational Unified Process (RUP): A software development process that is use-case driven, architecture-centric, iterative, and incremental.

regression test: The process of verifying that existing functionality in prior builds still works when new functionality is added.

stress test: A test to measure response times and system resource utilization of system under load limits.

test plan: A written document detailing the plan that describes your organization's testing strategies, resources, and schedule.

transition phase: The fourth phase of the Unified Process software life cycle. During this phase, the software is turned over to the user community.

Unified Modeling Language (UML): A standard modeling language for software systems.

use case: System functionality that provides value to some user, whether that user is a human or another system. Used to capture functional requirements.

workflow: A process that realizes a part of a business use case.
Appendix B: Tools

This section documents various tools that can be used to implement the processes discussed throughout this document.

Design (use case analysis)

Use the following tools:

- Embarcadero GDPro
- Microsoft Visio2000
- Rational Rose
- TogetherJ.

Test case management

Use the following tools:

- Mercury Test Director
- Rational Test Manager
- Compuware’s QADirector.

Code testing tools

The code testing activity lends itself well to test automation techniques, of which one tool has proven particularly useful and popular as of late. JUnit is an opensource test coverage tool freely available from the JUnit site. This tool is useful for developing scripts, which can be used to perform automated tests for incorporation in the build process using another freely available tool, ANT. There are many extensions that even perform testing at the Web container interface. The tool allows the developer to create test suites that can be aggregated into larger suites based upon testing dependency.

Automated Web testing tools

One tool that is invaluable throughout the testing workflow is the Web application load-testing tool. This tool provides the ability to create scripts to go through JSP/HTML pages and verify resulting content in an automated fashion. It also supports the ability to create load test using virtual users and to simulate expected system load and resource coverage. There is even a free Web-testing tool available from the Jakarta project, called Apache JMeter. It is not as powerful and complete as a commercial tool like LoadRunner,
but if you are limited by budget, JMeter is far better than no tool at all. No matter which
tool you choose, master it!

Note: Development organizations cannot overemphasize the importance of the
role associated with creating and maintaining the Web testing scripts.

Some of the commercial product vendors include:

- Compuware
- Empirix/RSW
- Mercury
- Rational
- Segue.

**Version control**

Use the following tools:

- Microsoft Visual SourceSafe
- Merant PVCS
- Rational ClearCase.

**Defect tracking**

Use the following tools:

- Elsinore Visual Intercept
- Merant PVCS Defect Tracker
- Rational ClearQuest.
Appendix C: Topics for further consideration

Due to the broad sphere of this subject, there are a number of items that warrant further elaboration. Here are some of the topics to watch for in future releases of this paper.

Case Study

Provide case study to illustrate the RUP testing workflow and supporting artifacts (e.g. use cases, test cases, JUnit samples, web-testing scripts, etc.).

Application Profiling

This section will cover the topic of application profiling. In context of the current document structure, this topic would fall under the construction phase’s code testing section.
Bibliography


References

Redbooks
SG24-6176-00  IBM WebSphere V4.0 Advanced Edition Handbook

On the Web
More information about Jakarta Ant can be found at http://jakarta.apache.org/ant/.
More information about JUnit can be found at http://www.junit.org/
IBM training course on WebSphere performance testing.
WebSphere Developer Domain Library: “Application Quality Assurance: Unit Testing”.

White papers
Best Practices: WebSphere Application Server Operations  by Peter Van Sickel
Best Practices: WebSphere Application Development Infrastructure by Mathew A. Oberlin
Acknowledgments

Wayne Beaton
Keys Botzum
Mike Capern
David Dhuyvetter
Bill Hines
Paul Kraur
Craig Lamprecht
Margaret Lisowska
Alexandre Polozoff
Peter Van Sickel
Contact information

To learn more about services, visit our website at http://www.ibm.com/websphere/developer/services.

To engage us, please contact one of our IBM Software Services Sales Specialists listed at http://www.ibm.com/websphere/developer/services/contacts.html.
Legal notices

Information in this paper was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

The information contained in this document has not been submitted to any formal IBM test as is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.