Sample Document

* Convert this document to the standard CT format.
* Convert all the headings, body text, images as per the CT Styles.
* Create a Table of Contents
* Check document for language and grammar.
* Time Duration is 30 minutes.

*This document contains Confidential and Proprietary Information of XXX, Inc.*

Revision History

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Date** | **Version** | **Change Description** | **Author** | **Reviewer** | **Approver** |
| 06/19/2012 | 1.0 | Initial Draft | Manoj Joshi |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

# Introduction

This section details out the purpose, scope of the document and Risk & Contingencies of the project.

## Purpose

The purpose of this Test Strategy is to define the overall approach that should be taken by the test team when delivering Testing services of the projects.

The Test Strategy presents the recommended approach to the testing of the target-of-test. For each type of test, a description of the test is provided and why it is being implemented and executed.

The main consideations for the Test Strategy are the techniques to be used and the criterion for knowing when the testing is completed. In addition to the considerations provided for each test below, testing should only be executed using known, controlled databases in secured environments. Following are the major business transactions which will result into test scenarios.

* Installing JBOSS components that includes copying updated files in the appropriate folders in JBOSS Machine
* Manual/visual validation of JBOSS and NOTES logs in order to check the successful deployment and business transactions.
* Manual/visual validation of appropriate error handling for invalid, missing, corrupted, not supported or repeating data during the execution for various UI configurations as well as configurations present in the file system.
* Manual/visual validation of appropriate error handling for invalid, missing, corrupted, not supported or repeating data during the execution for various UI fields.
* Manual/visual validation of the business transactions at the backend as well as on the User Interface.
* Validation of accurate data transmission from the end point reference which invokes the web service and accepts the response as well as payload back.
* Validation of accurate data transmission from the UI to the database.

## Scope

This document provides a detailed overview of the methods of testing, including product and project risk management, the division of testing into levels, or phases, and the high level activities associated with the testing. It also helps to clarify the various testing phases, Release build control and versioning.

## Risk and Contingencies

|  |  |  |
| --- | --- | --- |
| **Risk** | **Severity** | **Contingency Plan** |
| Availability of SUT(System under Test) on time | High | Reduce the number of test cases without affecting the ability to test the application thoroughly.(Covering critical business scenarios)  Move the testing schedule behind by a suitable number of days to accommodate the delay. |
| Delays in the turnaround time for fixing critical bugs. | High | Strong management of bug resolution would be required from Development to ensure bugs are fixed and available for re-testing in the scheduled time. |
| Poor code quality | High | Need to work closely with the development team to help improve the quality of dev testing. |
| Testing staff shortages / Unavailability | Medium | Plan to have buffer resources to cover for such situations.  Have the option of recruiting additional staff.  Consider the planned leaves during estimations  Reduce the number of test cases. |
| Latency accessing test environments or test environment not accessible  (configurations in the Lab Manager) | Medium | During estimation plan to have buffer time against such risks |

# Testing Approach

Following are the different types of testing that would be performed by various stake holders in order to deliver the quality of software.

## Integration Testing

Integration testing is a progressive testing, where modules at component level are added in a hierarchy and tested as a new component gets added. It assures robustness of the system when any integration happens.

QA will be responsible for to create and execute the integration tests. As the various components/modules/functionality gets released to the QA, it is expected that the test coverage will also be having the integration tests. All databases should be loaded with a set of test data so that all logic can be initially tested.

|  |  |
| --- | --- |
| Test Objective: | Objective of Integration testing is to assure interaction of two or more components produces results that satisfy functional requirement. |
| Technique: | Integration test cases are executed which are developed with the express purpose of exercising the interface between the components. |
| Completion Criteria: | Integration testing is complete when testing team makes sure that all the interfaces where components interact with each other are covered. |
| Special Considerations: | It is important to cover negative cases as well because components might make assumption with respect to the data. |

## System Testing

System testing covers following:

* Functional testing
* Non-functional testing

System testing covers end to end functionality as per the business requirement. In non-functional testing Performance and Security testing are included. System testing will include many separate tests to ensure that the application will work as expected in production. This will be in an environment that mirrors the ultimate production environment. As many tests as possible will be conducted with a subset of our users, so that we can simulate the actual Web transactions going from our application to the database.

|  |  |
| --- | --- |
| Test Objective: | Objective of Integration testing is to assure the functional and non-functional requirement from the production or the business case perspective. |
| Technique: | Functional manual testing is to be carried out. Non-functional testing is to be carried out with the help of automation tools. Refer section [4.4](#_Automation_tools) from the same document for more details |
| Completion Criteria: | Successful execution of all the test cases and closure on all high and medium severity defects. |
| Special Considerations: | There could be few defects which could be due to the environmental conditions. In this case, the same these should be captured in the Release notes document as the known Issues. |

## Acceptance Testing

For all intents and purposes, the system will be placed in a production status, including hardware and software. QA will coordinate testing with the Product/Dev Managers to ensure that the resulting transactions are as they expect.

# Test Specification

## Testing environment structure

### Development

This is the common environment where all developers commit code changes. The goal of this environment is to combine and validate the work of the entire project team so it can be tested before being promoted to the QA Environment.

* **Limited subset of data that is useful for testing “boundary conditions” in the application. It may be wise to refresh this subset of data frequently to remove the artifacts of software development and testing on the Integration environment.**

### QA

The purpose of the QA environment is to simulate as much configuration of the Production environment as possible. QA should be able to replicate all processes clients will follow from Installation to actual use of application.

* **Identical software configuration as the production machine** and a **complete, independent copy of the production database** so it is a true basis for QA testing.
* **Comparable hardware configuration to the production system** so an accurate forecast of capacity by performance testing against it and then multiplying its performance by the number of machines that will be deployed in production.

### Staging

The Staging tier is an environment that is as identical to the production environment as possible. This should have same release version as on Production. The Staging environment can also double as a Demonstration/Training environment.

* **The same limited subset of data as the QA environment.**

### Production

The production tier might include a single machine or a huge cluster comprising many machines.

These tiers speak of “environments” rather than “machines” or “servers.” The Production environment should be by itself and not shared with any of the other environments.

## Requirements for Test Environments:

### Requirements for Committing Code to Development Environment:

In narrative form, the software developer writes code in his or her local environment and checks it into the Subversion source code repository after unit testing was completed. As other developers report bugs more changes are made and checked in. All new features and bug fixes should be integrated and tested in Development environment.

### Requirements for Installing a Release on QA Environment:

When the developers are happy with the behavior of the Development environment, the Release Master creates a Release Candidate or “tag” of the code in Subversion for QA. QA installed the release candidate on the QA environment. At this point the quality assurance (QA) testers start their review. The QA area also doubles as a training environment when the Production release is ready. QA reports go back to the developer who fixes them and checks the changes into Subversion. After all of the bugs are fixed, the QA manager promotes a new version to QA.

### Requirements for Promoting a Release to Production:

Testing process continues until the QA team declares the staging version is “okay to release”. The release manager packages up the release version from Subversion and deploys it on the Production and Staging servers. As time goes on, bug reports and feature requests are made for which the developer writes code and checks in the changes to the source code repository. Repeat until the end user is completely satisfied.

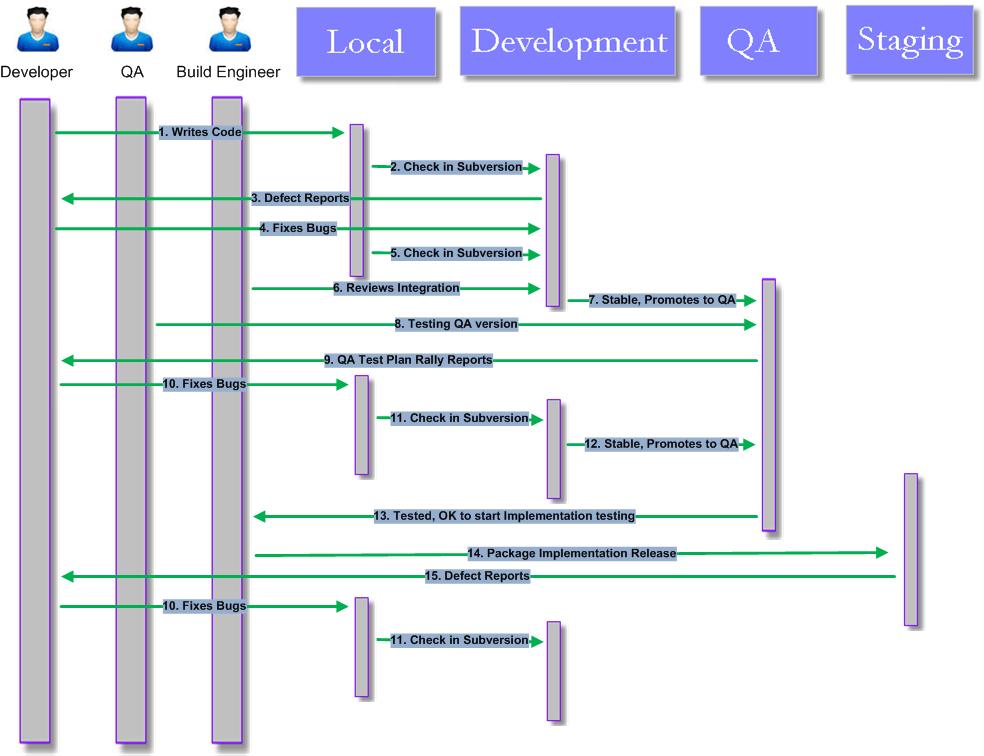


Figure 1

## Important Notes

**Developers only make changes to the Development and QA environments.** If a bug fix is to be made, the developer makes it in Subversion at the QA stage.

In order to maintain the integrity of the source code repository *at no point does a developer make changes directly to the Staging or Production environments.*

**For each deployment to Production, there are multiple versions in Staging** and **for each deployment into Staging; there are multiple versions in Development/QA**. By design, end users are isolated from the rapid and occasionally buggy process of developing software. It is assumed that most bugs will be caught early and repeated versions at the early stages fill find bugs faster.

**Only “release manager” can deploy versions to the next stage.** There can be different release managers for deployment from QA-to-Staging and Staging-to-Production; the release manager can even change from version to version. The important point, though, is that there is always only one person responsible for deploying the new version. *Nothing goes in QA, staging or production environment without QA manager approval.*

Instead **the version of software that is promoted to the Staging environment is “tagged” off of “trunk” in the source code repository** and it is the tagged version that is copied to Staging. Work by the developers then continues on the “trunk”. The same holds true for the promotion of the Staging version to production.

# Test Scripts

Testing the functionality available in the application will be accomplished in the following manner:

* Validation of data input and display in each available field in the User Interface
* Validate the usability of the User Interface (UI) from the end-user perspective
* Validation of UI flow and functionality using a combination of input devices
* Validation of backend flow and functionality using a combination of input devices
* Validation of Error log data generated from all valid UI input
* Validation of appropriate error handling of invalid or missing data for all required input fields in the UI
* Validation of appropriate error recovery (executing appropriate negative cases)
* Validation that the UI generated data is accurately committed to the appropriate database tables
* Validate database for the data sent from UI.
* Validate the Pentaho Job is Up and Running.
* Validate the CCD generated from the ACS Connector for various scenarios.
* Validate the output generated for CCD using modified VHR xsl.
* Validate CCD in VHR received from ACS connector.

Several concerns affect the specific test procedure to be used, including:

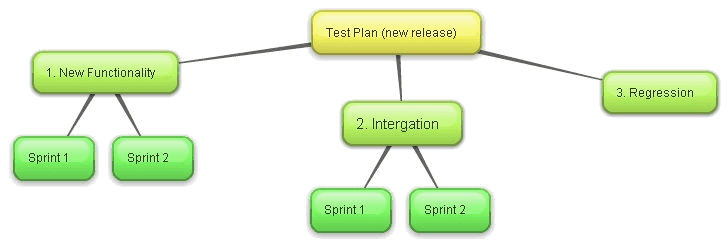
* **Dependencies among individual test cases.** Some test cases can only be run after others because one is used to set up another. This is particularly true for systems that operate continuously, where the later test run may need to start at a state set up by the earlier one.
* **Defect detection related sequencing.** Many problems can only be effectively detected after others have been discovered and fixed. For example, integration of several components and related testing typically focus on interface and interaction problems, which can be masked by problems in individual components. Therefore, these components need to be individually tested before integration testing starts.
* **Sequences to avoid accidents.** For some systems, possibly severe problems and damages may incur during testing if certain areas were not checked through related test runs prior to the current test run. For example, in embedded software for safety critical systems, one does not want to start testing safety features before testing other related functions first. This can be considered as a special case of the problem or defect related sequencing where there is a very strong economical incentive for preferring certain sequencing to others.
* **Problem diagnosis related sequencing.** Some execution problems observed during testing may involve complicated scenarios and many possible sources of problems. Under this situation, related test runs focused on a single aspect or limited areas can be used to help with the problem diagnosis. Better yet, if such complicated problems are expected, we should run related simpler test cases first to eliminate certain possibilities and narrow down the problem areas. Therefore, one natural sequence for test case execution commonly used in practical testing procedures is to progress from simple and easy ones to complicate and difficult ones. The same idea will be used in defining coverage hierarchies.
* **Natural grouping of test cases**, such as by functional and structural areas or by usage frequencies, can also be used for test sequencing and to manage parallel testing. However, among areas where no such order exists, or when the incentive for following a certain order is not strong, we can carry out testing for them in parallel to speed up the testing process.

## Test Scripts: Functional Area UI & Database

Starting at the “Home” page, navigate to the following pages and test input of valid data ranges in the listed input fields (shown as “<fieldname>” where input is required in an alpha-numeric field with “(o)” denoting that the field is Optional) for each page/function (buttons, actions, and fixed dropdown selections are shown in “ALL CAPS”)

## Pre requirements to run test cases

All the test cases are uploaded in Rally with the per-defined Test folder. The folder structure is as follows:



All the test case should explicitly capture the pre-conditions and the tear down conditions.

# Test automation

Test automation aims to automate some manual tasks with the use of some software tools.

The demand for test automation is strong, because purely manual testing from start to finish can be tedious and error-prone.

The key in the use of test automation to relieve people of tedious and repetitive tasks and to improve overall testing productivity is to first examine what is possible, feasible, and economical, and then to set the right expectations and goals. Currently, there are no test cases automated. However it is one of the target areas in future. The following sections are captured for the future reference.

## Test execution

Among the three major test activities, preparation, execution, and follow-up, execution is a prime candidate for automation. In fact, this is the area in which the earliest test automation tools found some unequivocal successes. For example, various semi-automatic debugging tools or debuggers allow testers to set and reset variable values and execution states during execution and observe the dynamic execution behavior at different observation points. These tools are semi-automatic because testers are still involved in test execution intervention. An additional functionality for many of the test automation tools is to allow information recording and collection.

## Test planning and preparation

Test case preparation is the area where there is some realistic potential for automation.

For example, in testing of legacy products, various automated analysis can be performed to compare the current version of the product with its previous versions, and to screen the existing test suites to select the ones for regression testing. For construction of new test cases, automation is also possible.

## Test measurement, analysis, and follow-up

In terms of analyses of test results and follow-up actions, the situation is similar to test planning and preparation. Most of the follow-up actions involve problem fixing and various other remedial and improvement initiatives, very little of which can be automated. However, specific analysis activities can be supported by various analysis and modeling tools. A general tools support strategy for QA and development process measurement, data analysis, and follow-up.

## Automation tools

### Database Testing:

* IBM DB2 Command Line-SQL

### Performance Testing:

* JMETER (Not yet implemented)
* HTTPWatch
* SoapUI (version above 4.0)

# Test Documentation

## Documents across test life cycle

### Test Specification:

Test Strategy: This defines the overall approach taken by the test team when delivering testing services to all of the related areas.

Test Suite: This converts the test conditions into test cases by adding real data, pre-conditions and expected results. Rally is used as the Test case management tool, where all the test cases are uploaded. The test folders are created in the predefined order. For more details refer section [4.2](#_Pre_requirements_to). From the traceability perspective the test cases majorly includes following:

* Work product which is mapped with the User story/Defect against which test cases are created and
* Test folder which is mapped with the Test plan from the Rally.

QA Sign Off: Once the regression testing is completed and depending upon the Exit criteria, QA will send the sign off document to the various stake holders of the project.

### Test Execution and Reporting:

All the information related execution status can be easily pulled up from Rally with the different menu options available. Some of the standard features for the test execution and reporting point of view are captured as follows:

* Quality – Test plan: which contains all the test folders
* Quality – Defect Summary: this gives release wise Defect Summary matrix
* Quality – Defects – Found in Current Iteration/Release