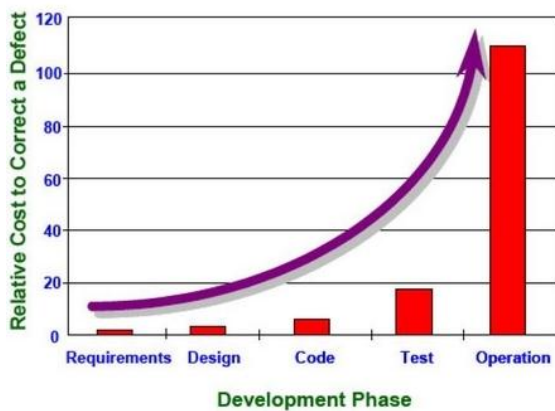


# Quality Assurance

## Introduction

Whether your project is a small change to an existing system or a huge first release of custom built software for a new business, Quality Assurance (QA) is critical to project success. QA is more than just testing. It is an integral part of the software development life cycle (SDLC) and actually begins when the requirements are being gathered, written and documented. The QA team participates in all phases of the development life cycle. In so doing, they can begin to write test cases, as well as validate designs. Professional QA teams also ensure that the code is being written to the expected standards and that it meets requirements such as page loading speed and performance scalability and maintainability. In MIS 374 we have emphasized the role of QA within the SDLC and the value of including QA in your project charter. The earlier errors are found, the cheaper they are to fix, as illustrated in Figure 1 below. Creating test plans early and executing them throughout the SDLC distinguishes a quality development organization from the rest.

Figure1. Defect Correction Cost Lowest if Found Early



Source: <http://www.gridshore.nl/2008/12/30/defects-lean-software-development-offshore-oh-my/>

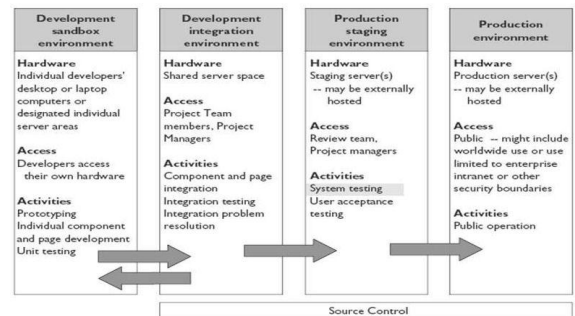
## Creating a Test Plan

Your project charter is the basis for your test plan. A good plan includes testing by application

developers and as many stakeholders as possible. The web flow diagram in your charter might be the best resource for creating a list of what needs to be tested and who will do the testing. Or, your functional and non-functional requirements lists could be the basis for your plan, depending on which documents provide the best overview of your system.

Your test plan should cover the unit testing of individual modules and pages in your development sandbox, as shown below in Figure 2. Integrated testing should occur in a different area, so programmers can continue to work with their code fixes without confusing the integrated testing process. The number of staging environments in your test plan will depend on the complexity of your system and your client's resources and project schedule.

Figure 2. Software Staging Environments



## Unit Testing

The "units" in unit testing are meaningful code modules or pages, such as "Login" and "Customer Profile." Unit testing is comprised of white box and black box testing. White box testing "opens the box" to look at what is going on inside. For example, if one application developer looks at the code, line-by-line for a new Customer Profile, that is white box testing. If an application developer or user enters Customer Profile data and then checks the results, without looking at the code, that's black box testing.

For black box testing, a test specification should be written for each unit of code using a form like

the one in Figure 3. The test cases should cover all your functional requirements, including error handling and data validation. A programmer should never write her/his own test plan because they will tend to focus on testing the things they know work correctly. A project manager, business analyst or another programmer should therefore write the test plan. Then, errors in design, as well as coding will be found and testing will be comprehensive.

A great time saver for generating data for unit testing is GenerateData.com. This same data can be used in integrated testing and system testing.

**Figure 3. Test Specification for Unit Testing**

Test Specification for < System Name here >		Page ___ of ___			
Designed by:		Module or Screen: -- Fill in Unit being tested here. --			
Test Data Source:		Objectives:			
Test Case #	Description	Test Steps	Expected Results	Actual Results	Performed by / Date

## Integrated Testing

In MIS 333k your team's development sandboxes were probably your own laptops. When you tested the code for several team members on Classweb2 to see if the pages worked together, you were doing integrated testing. Classweb2 became your "Development Integration Environment," using the IT professional terminology in Figure 2. The best integrated testing is done by a business analyst or project manager who knows the requirements and scrutinizes every detail of the output. If your entire team is coding as programmer-analysts, then be sure and trade roles. For example, the programmer-analyst who developed the report functionality should test the customer-facing pages or some other area that is not part of the programmer-turned-tester's drill down area of expertise.

## User Testing in the Build Phase

Users are your best testers. Show them parts of the system as early as possible. Give them access to an area of the development sandbox so they can add data and maneuver in the unfinished pages while you are not present to explain messages and events. Helpful users will email problems and questions, thereby saving you trouble with errors that persist in the system through integrated testing. This early user testing is a test of your understanding of the requirements, as well as a test of the clarity of your design and correctness of your code. The sooner these errors are found, the less trouble and expense of fixing them, as illustrated in Figure 1.

User testing also serves as a test of on-line help, quick reference guides, and other user materials, so write first drafts early and ask users to improve these as part of their black box user testing.

## Regression Testing

As changes are made to the system, retest the functionality with the same tests. Changes can create errors, so testers need to *regress* to the old tests to make sure new errors have not been introduced.

Regression testing is so important that automated test environments are created to rerun test scripts and save significant time and resources.

## Early Testing of Purchased Software and Open Source Solutions

Too often developers assume that purchased software or open source solutions will work as promised. At an early stage—as part of the evaluation of software tools in the Inception Phase—run black box tests to be sure the needed functionality works. Once the software is determined to meet the client's needs, treat the black box testing in the Build Construction iterations just as seriously as you would for custom built software. System errors can be caused by misunderstanding functional requirements as easily as by code errors.

## System Testing

Once the functional requirements have been tested as integrated modules, the new software must be tested as a complete system. The system test brings together the trained users, the reference and user materials, several test data files, and the software and hardware to determine how the components perform as a single functioning unit. System tests are often performed after hours—in the evening or over one or more weekends. Ideally, a production staging environment will be created that is very similar to the production environment, as shown in Figure 2. If the first real tests of the production environment do not occur until the actual installation of the software in the production environment, this last system test is likely to be a lengthy and stressful process.

The system is put through several test runs. One test run may require the users to enter a small set of standard production data to verify that the data entry screens are easy to read, that the procedures for recovering from data entry errors are clear and correct and that the system produces the expected output.

Another, more extensive test run, might test the system's capacity—will it handle the expected peak production volume? One or more of these tests should be a walkthrough with several users. How well does the system meet specifications for response time? Can it process the expected data volumes as quickly as required? What are its storage, throughput, and output limits? When pressed past its limits, does the system degrade gracefully or does it crash?

Yet another test run might introduce intentional hardware failures to test the backup and recovery procedures. Is the audit trail accurate? Have transactions been logged correctly? Do the users understand how to perform the necessary manual backup procedures? Can they restart the system without losing data or duplicating transactions? Once these controlled tests have been completed and the results meet user acceptance criteria the system is ready for installation.

Even after the system has been accepted by the users, testing is not usually complete until the

system has been used for day-to-day production operations. Systems that have passed all the tests mentioned above have been known to collapse during their initial production runs. Such a breakdown is particularly disastrous in a real-time environment such as airline reservations or consumer banking.

## Automated Testing

Testing is tedious work, especially for graphical user interfaces. Capturing the key strokes for a series of tests and then rerunning the scripts—rather than reentering the tests manually—saves time and increases the chance that all the necessary tests will be run after changes to the code. For systems with multiple releases, this is a huge time saver as well as a way to decrease the risk of introducing errors into new releases. A variety of vendors sell test tools. These testing processes are a standard part of the Development Integration Staging Environment illustrated in Figure 2.

## User Acceptance Testing

Once all components have been installed, the system is subjected to a final user review—the acceptance test. During the acceptance test, users test the system under routine and exceptional conditions to determine whether it satisfies the acceptance criteria that were established by the Project Charter. Users scrutinize documentation; measure system response times; study input screens and reports; evaluate backup, recovery, and security procedures; and rate the system's usability and reliability. Any components or functions that fail to satisfy the acceptance criteria must be modified. In other words, the development team is not off the hook until the users signal their acceptance of the new system by signing off on each criterion. Figure 4 is an example of a user acceptance review form.

**Figure 4. User Acceptance Review Form**

<b>User Acceptance Review Form</b>		Date	
System		Review Date	
Review Prepared by			
Acceptance Criteria	Approved by	Date	
1.			
2.			
3.			
4.			
5.			
6.			
7.			
8.			
9.			
10.			
<b>Acceptance Acknowledgement</b> The undersigned representatives agree that this project, identified above, has been completed in a satisfactory manner.			
<u>Information Systems Management</u>		<u>User Management</u>	
Authorized Signature	Date	Authorized Signature	Date
_____	_____	_____	_____
Title		Title	

## Tips for Complete Testing

- Have users test to ensure that the functionality they require is there and that they understand the navigation, messages, on-line help, and other user materials.
- Avoid “developer fixation;” be sure to ask business analysts and project managers to test the code. Or ensure that programmer-analysts test each other’s code, so that the programmer who knows an area’s detailed requirements, designed and coded the functionality, isn’t just testing what was in his or her own view of that function.
- Test each function in the system separately—unit testing.
- Write at least one test for every functional requirement, including all data validation requirements. This includes checking for

the wrong kind of data, for example, a negative number of employees in a payroll program or entering numbers where letters are expected.

- Check the flow of control through the entire system to ensure that the links work: that all desired paths exist and modules communicate correctly with one another. This is integrated testing.
- Test valid, invalid, and boundary values. This includes simple boundary checks, such as minimum, maximum, and off-by-one values. Verify that all error-detection routines work as expected.
- Test compound boundaries that is, combinations of input data that might result in a computed variable that’s too small or too large.
- Test data that is not supposed to change for consistency.
- Test system backup and disaster recovery procedures.
- Test for compatibility with old data.

Test for browser compatibility, especially any older versions that are still common. (If the software does not work for older browsers or operating systems, include a test for the older products and write a clear error message for users.)

## Templates on Resources Page

- Test Specification Template
- User Acceptance Template

## Examples on Resources Page

- Waldorf School Test Plan and Specifications (Web site)
- Latinitas Test Specs
- Judy Paul.com Test Specs
- Test Script example