Smarter Balanced Assessment Consortium Recommendation

Smarter Balanced Quality Assurance Approach

Recommendation for the Smarter Balanced Assessment Consortium

20 July 2012

Summary

When this document was originally proposed, Measured Progress was to provide support and expertise in developing the overall testing strategy for the assessment system, review and provide feedback on quality assurance (QA) testing activities, and support the efforts to determine whether the testing had been successful. Measured Progress discussed with the consortium the need to cover the overall assurance of the system required for a project of this type.

Since that time, much of this work has been consolidated into the Test Delivery work (RFP-11). In light of this addition, and to avoid unnecessary duplication with that effort, this document focuses on giving guidance to the consortium on the types of challenges they will face with respect to QA, and recommendations on how to best approach those challenges. This document, while not a testing strategy itself, presents a case for the importance of a testing strategy. This document outlines some of the key components of a good testing strategy and makes recommendations for the Smarter Balanced assessment system.

Measured Progress will still provide support and expertise in developing the overall testing strategy for the assessment system and review and provide feedback on quality assurance (QA) testing activities and support the efforts to determine whether the testing efforts had been successful.

Background

In this industry and with a system of this highly visible nature, the effects of software that has not been sufficiently tested can lead to an array of problems during a test administration that can be financially and politically expensive. Almost all of the high profile mistakes that have been made in the past decade with online assessments could have been avoided with a more complete quality assurance process. With complex systems, stating that QA was done is not sufficient. The quality of a piece of software starts with the development team. Testing must be planned for in advance and not tacked on at the end of a project. As with any software development project, it is important to have a clear testing strategy in place and a commitment to that strategy from everyone involved. A good testing strategy will help:

- Ensure that all components meet the defined requirements and function as expected.
- Minimize the risk of introducing major bugs to the operational system.
- Ensure that performance issues are caught in time to fix them.
- Confirm that all components are successfully integrated.
- Ensure that applications are compatible and function correctly on all supported configurations.
- Identify Issues early in the process so they are less expensive to fix.
- Minimize the chance of major incidents once the system goes live.
- Give confidence that the system can handle the expected traffic.

Recommended Approach

A test strategy is a high level and overarching document that outlines the structure of all QA activities. It is important to note that all software projects have a test strategy either formally or informally. A formal strategy will give all of those involved with the project greater visibility as to the state of the system, a more accurate picture of the barriers that lie ahead, and a better chance of meeting the objective. The strategy should be developed in coordination with the development teams, but is owned by the group in charge of the overall assurance of the system. The following is a list of guidelines that should be considered in the Smarter Balanced Test Strategy.

- Recognize Efficiencies: Don't add process unless it adds real value.
- Clear Responsibilities: All roles should be clearly defined, and overlap in responsibilities should be minimized.
- Definition of Done: Testing is never complete, so you need a way to determine when a component or a system is ready. Ensure that clear definitions of "done" are in place for all releases.
- Entry and Exit Criteria: Establish entry and exit criteria for all environments.
- Quality from the Beginning: Incentivize all team members to think about quality and get the development team to take ownership.
- Common Tools: Having a common toolbox and testing framework is strongly recommended. This includes defect trackers, continuous integration, testing frameworks, and automation tools.
- Common Language: Ensure that key terms and concepts for the assurance process are clearly defined so that all teams are using a common vocabulary.
- Quality Metrics: Standardized metrics should be used to help determine the quality of each component. These metrics should be compiled for each sprint to enable the viewing of trends.
- Define QA Artifacts: Establish guidelines for unit tests, test coverage, test cases, defect attributes, etc.
- Platform Compatibility Testing: A strategy needs to describe how to test all supported devices, operating systems, and browsers.
- Integration Testing: A strategy needs to describe how to verify that components are successfully integrated.
- Performance Testing: A strategy needs to describe how to verify, with defined metrics, that the system can handle the anticipated load and still be responsive to end-users.
- Security Testing: A strategy needs to describe how to verify that sensitive data is being adequately secured both at rest and in transport between components.

Roles and Responsibilities

With many teams involved it will be important to clearly define who is responsible for each assurance activity. This section proposes the key testing responsibilities that will be undertaken by each group involved with this effort. These can be refined with the Architecture Core Team and should be approved by the Architecture Review Board.

Development Vendors

The development vendors would be those responsible for actually developing the application components for the assessment system. There will likely be multiple vendors on this project and each should have their own QA staff that works closely with the developers and performs the bulk of testing activities required.

Responsibilities

1. Ensure that staff has the skills necessary to successfully perform testing activities.
2. Maintain all environments needed for testing.
3. Manage a defect-tracking database and keep it up-to-date.
4. Develop test plans.
5. Create and maintain test cases and automated test scripts.
6. Execute test cases and generate test summary reports.
7. Generate traceability reports.
8. Facilitate User Acceptance Testing.
9. Perform system, integration, regression, compatibility, and load testing.

Architecture Core Team

The Architecture Core Team is comprised of the lead architects from the development vendors that have been selected to develop and deploy Smarter Balanced architecture components, as well as the Measured Progress Enterprise Architect.

Responsibilities

1. Review and approve all test plans.
2. Establish best practices to be followed by all testing teams.
3. Resolve disputes that arise from integration testing.

4. Define test summary report standards.
5. Review and approve all test summary reports.
6. Record and document any risks that arise from the testing process.

Architecture Review Board

The Architecture Review Board (ARB) will oversee the testing process and serve as a reviewer of the testing artifacts. The ARB reviews and provides actionable feedback from the testing process that will be reported back to the other teams.

Responsibilities

1. Accept or reject any testing standards used.
2. Accept or reject test summary reports.
3. Provide guidance with any integration issues.
4. Record and document any risks that arise from the testing process.
5. Assist with user acceptance testing.
6. Accept or reject changes to the testing strategy itself.
7. Accept or reject quality metrics.

Smarter Balanced Steering Committee and Work Groups

The Smarter Balanced Steering Committee provides the overall management and reporting for the Smarter Balanced architecture. The work groups are responsible for the development of the assessment system. These groups specifically:

- Define specific scope and timeline of the work group to accomplish the deliverables and milestones identified in the Smarter Balanced Master Work Plan.
- Develop work plan and resource requirements to guide vendor and work group activities.
- Determine and monitor allocated budget.
- Oversee and direct Consortium work in assigned content area and oversee and direct vendors, if applicable.

Responsibilities

1. Ensure development vendors are coordinating during integration process.
2. Work with development vendors and Architecture Core Team to mitigate risks.
3. Conduct user acceptance testing.
4. Sign-off on test plans.
5. Sign-off on test summary report requirements.
6. Sign-off on quality metrics used.

Key Concepts

The following section provides a review of key Quality Assurance terms and processes. This section is not exhaustive and is only intended to provide descriptions for the types of testing and key concepts that will be encountered with this effort for those less familiar with QA.

Testing Modalities

The following are descriptions of the types of testing associated with QA.

White Box

White Box testing is a testing method typically performed by a person with an understanding of the application's code structure that tests the internal workings of an application. For example, an automation engineer will write scripts to verify API functionality, or a manual tester might utilize SQL to query database insertion results.

Black Box

Black Box testing is conducted without any knowledge of the underlying code and tests the functionality of an application. This can be accomplished through manually testing the user interface or by utilizing automation scripts to step through the application.

Unit

A White Box testing approach tests the smallest testable parts of an application. Unit tests can be conducted manually but are usually done by writing code that can be run to test individual functions and entire modules of an application. Unit tests are helpful for ensuring that new changes made to the application do not break existing functionality, and are typically a prerequisite to the build process.

Smoke

A smoke test is used to determine the operability of a build or new installation. It typically contains a small subset of test cases that verify high-level functionality such as login, page transitions, page objects, and a very small piece of functionality. In a continuous integration environment, automated smoke tests are attached to the CI process and run automatically after the build has been installed. Smoke tests can also be performed manually.

Functional

Functional testing may be performed manually or with automation. It can also be referred to as Black Box Testing (see above). Functional testing is performed against pre-defined test cases and verifies that the features are performing as expected. Functional testing includes, but is not limited to:

- Verifying application functionality against stories/specifications.

- Verifying user interface controls.
- Verifying negative test cases. This is a very important aspect of functional testing which can't be ignored.
- Verifying data transmission to/from the data store.

Integration

After a component has successfully completed unit testing, the component is combined with other components and tested as a group. This type of testing ensures that component interfaces are functioning as expected, and that no functional or performance issues have been introduced when connecting multiple components.

Integration testing is often performed in a large enterprise environment where multiple applications combine to form a single deliverable. It can also be utilized in an environment where unit tests are brought together to verify a system. Oftentimes this is performed at the automated or manual functional level, depending on pre-defined project processes.

User Acceptance

User acceptance testing is a process step where the end user (internal or external) is provided access to the system to verify that the requested functionality performs as defined. This is not an opportunity for end-users to add new functionality or make changes. Traditionally, this is performed later in the development process. However, it is highly recommended that user acceptance testing be performed as early and often in the process as possible.

Automation

Automation adds efficiency to the testing process by leveraging test automation tools that can quickly run through test cases and report any unexpected results. Automated test cases can be created using scripts that interact directly with the component interfaces (UI or Web Service) and compare the actual results with the expected or predicted result. An example test case would be logging into the application with a blank password and checking for an "Invalid Username or Password" response. Uses for automation include, but are not limited to:

- Performing smoke tests for new builds and installs.
- Conducting functional testing during any step of the quality cycle.
- Performing regression testing.
- Creating utilities to assist with manual testing. (i.e. a scripted solution for quickly verifying database contents).
- Discovering process improvement (i.e., replacing human intervention for a tedious support task such as setting up data to prepare for a given manual test).

All automation must be architected to reduce maintenance through code re-use. Any given function that repeats throughout tests or applications should be stored in a single script file and called each time it is needed. By doing so, re-work is only needed in one place should underlying functionality change

Performance

Performance testing encapsulates a range of strategies used to verify the ability of the application being tested to perform under high users loads as defined by the client. This includes, but is not limited to:

- Application Performance Management (APM): This approach uses third party software to trace application requests through each line of code as the application is being used. The software then identifies bottlenecks based on length of code execution and other variable factors. APM tools are used in the development and QA environments and can be leveraged to monitor ongoing code performance and alert when negative changes have occurred.
- User interface response times: It is imperative that a UI responds as quickly as possible to the user. This will often be determined by the type of operation and is not always controllable (i.e., a complicated server-side database query). Response time performance testing seeks to record the time from the moment a user makes a request at the UI to the time the request is returned. Two approaches may be taken to determine request time. The first assumes that the transaction is complete as soon as the first packet has been returned from the server. The second approach does not record the return time until all objects and text are known to be loaded on the page. It is highly recommended that the second approach be utilized for maximum precision in regards to validating response times that directly correlate with a user experience.
- Bandwidth Testing: The primary function of bandwidth testing is to determine the performance of an application across varying bandwidths. This is an important test in the education environment as many schools and districts have varying degrees of network speed and connectivity. Bandwidth testing is performed using a third party application to emulate low and high bandwidths while a performance script is running.
- Load Testing: The ability for an application to support a high number of users must be verified through a load test. Load tests stress the underlying systems, including hardware, databases, code, web servers, load balancing, database servers, and other areas of a solution. The premise is to inject a large number of concurrent users into a system using a scripted solution.
- If a user interface is involved, it is highly recommended to write all tests in a headless manner. Headless is a term used to identify interaction with a system through system calls rather than direct user interface interaction. In this manner, more users can be driven from a single performance node (a server which is generating user load), thus reducing hardware or VM

requirements. Third party load testing tools can be used which range from high costs to free, effective, open software (The Driver).

Testing Artifacts

The following describes the key artifacts that will be generated during the QA effort for this project.

Test Plan

Test plans determine the overall approach for testing an application and outline the strategy, scope, team, risks, approaches to testing, and other pertinent information.

Test Case

Test cases are individual artifacts that determine the testing steps and expected outcome for a single piece of functionality. Multiple tests may not be included in one test case. Test cases provide an important basis for project reporting. A key performance indicator (KPI) for application quality can be determined by examining the percentage of defects found against test cases executed. Less than 5% is a positive indication.

Test Scripts

A test script, as a component of automation, is a block of code that represents automated testing for individual or multiple test cases.

Test Reports

Test reports should include at minimum:

- Number of test cases completed.
- Percentage of test cases completed.
- Number of test cases outstanding.
- Percentage of failed test cases against number executed.
- Number of bugs broken out by criticality.