ORCHESTRATOR

SOFTWARE QUALITY ASSURANCE PLAN

TM-SQA-01 V2.0

DECEMBER 1, 2015

Tecnológico de Monterrey

This page intentionally left blank.

ORCHESTRATOR

SOFTWARE QUALITY ASSURANCE PLAN

TM-SQA-01 V2.0

DECEMBER 1, 2015

SQA Plan Approvals:

Eric Flores	**19/11/2015**
SQA Manager	**Date**
Christian Espinoza	**19/11/2015**
Project Manager	**Date**
Francisco Gutierrez / Ivan Faria	**19/11/2015**
Program Manager	**Date**

RECORD OF CHANGES

*__A__ - ADDED __M__ - MODIFIED __D__ - DELETED

VERSION NUMBER	DATE	NUMBER OF FIGURE, TABLE OR PARAGRAPH	**A* M D**	TITLE OR BRIEF DESCRIPTION	CHANGE REQUEST NUMBER
1.1	19/11/15	Entire Document	A	Updated and filled template to include Challenges, Assumptions, Tactics and Integration Tactics Sections.	
1.2	25/11/15	Entire Document	A	Revised Acronyms, Purpose, scope, and management sections.	SQAPT-003 SQA-0001
1.3	29/11/15	Entire Document	A	Revised Risk Management and Challenges for Orchestrator sections. Incorporate a separate section for Acronyms and revised Management section.	
2.0	01/12/15	Entire Document	A	Added the Assumptions for Orchestrator QA Section. Added Integration Tactics for Orchestrator section.	

TABLE OF CONTENTS

Section **Page**

Contents

SECTION 1. PURPOSE

The purpose of this plan is to define the Orchestrator Software Quality Assurance (SQA) organization, SQA tasks and responsibilities; provide reference documents and guidelines to perform the SQA activities; provide the standards, practices and conventions used in carrying out SQA activities; and provide the tools, techniques, and methodologies to support SQA activities, and SQA reporting.

1.1 Scope

This plan establishes the SQA activities performed throughout the life cycle of the Orchestrator project.

The goal of the SQA program is to verify that all software and documentation to be delivered meet all technical requirements. The SQA procedures defined here shall be used to examine all deliverable software and documentation to determine compliance with technical and performance requirements.

Table 1-1 shows the software life cycle activities of the Configuration Items (CIs), as identified by the Institute of Electrical and Electronics Engineers (IEEE)/Electronic Industries Association (EIA) Standard 12207 Series, Software Life Cycle Process, reference (b), to which this SQA Plan applies.

1.2 System Overview

The TBTAF System which stands for Tag Based Test Automation Framework, it's a framework for automating testing. With this framework no brute force approach will be needed, just execute what it's needed based on what really got impacted on production code. Using documentation tags between production and test code will enable smarter automated testing.

The system includes Interpreter, Discoverer, Listener, Databridge, Executor, Orchestrator subsystems within the system. Figure 1-1 identifies each subsystem.

TABLE 1-1. SOFTWARE LIFECYCLE ACTIVITIES

SOFTWARE LIFECYCLE ACTIVITY
Project Planning and Oversight
Software Development Environment
System Requirements Analysis
System Design
Software Requirements Analysis
Software Design
Software Implementation and Unit Testing
Unit Integration and Testing

1.3 Document Overview

This document identifies the organizations and procedures to be used to perform activities related to the Orchestrator SQA program as specified in IEEE Std 730-1998, IEEE Standard for Software Quality Assurance Plans, reference (c) and IEEE Std 730.1-1995, IEEE Guide for SQA Planning, reference (d).

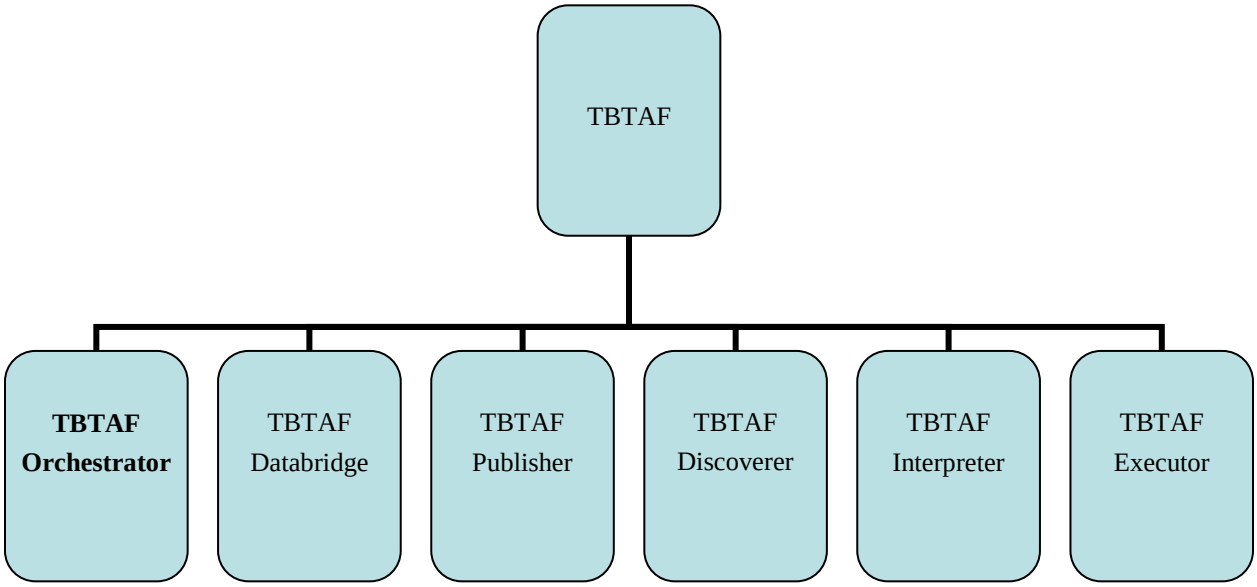


Figure 1-1. TBTAF System Software

1.4 Reference Documents

This section lists the documents referenced in this SQA Plan.

a. TBTAF Introduction Slides.
b. TBTAF Architecture Diagram.
c. https://github.com/S41nz/TBTAF/wiki/Orchestrator.

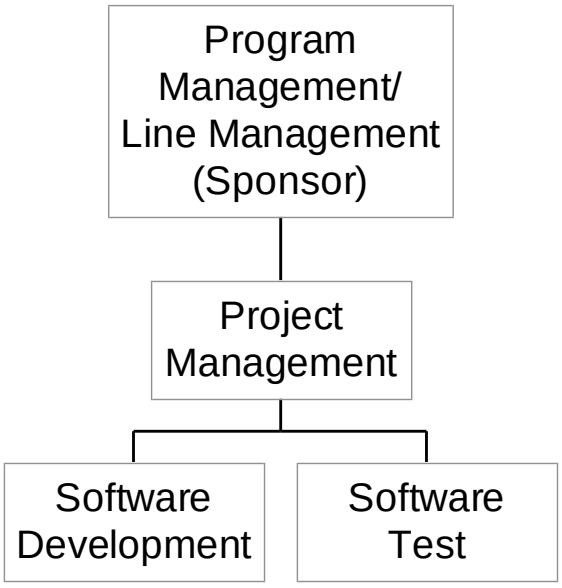
SECTION 2. MANAGEMENT

This section describes each major element of the organization that influences the quality of the software.

2.1 Organization

Good software practice requires a measure of independence for the SQA group. This independence provides a key strength to SQA; that is, SQA has the freedom, if the quality of the product is being jeopardized, to report this possibility directly above the level of the project. While in practice this rarely occurs, for almost all problems are correctly addressed at the project level, the fact that the SQA group can go above the project level gives it the ability to keep many of these problems at the project level.

Figure 2-1 shows the SQA organization with relation to the project organization.



SQA is responsible for ensuring compliance with SQA requirements as delineated in this SQA Plan. The SQA organization assures the quality of deliverable software and its documentation, non-deliverable software, and the engineering processes used to produce software.

The following describes the functional groups that influence and control software quality.

a. Program Management is responsible for the following items:
   1. Establishing a quality program by committing the project to implement the Software Engineering Process Policy, reference (g), and reference (a).
   2. Reviewing and approving the Orchestrator SQA Plan.
   3. Resolving and following-up on any quality issues raised by SQA.
   4. Identifying an individual or group independent from the project to audit and report on the project's SQA function.
   5. Identifying the quality factors to be implemented in the system and software.
   6. Fulfillment of Orchestrator SQA Activities.

b. Project Management is responsible for:
   1. Implementing the quality program in accordance with references (g) and (a).
   2. Identifying the SQA activities to be performed by SQA.
   3. Reviewing and approving the Orchestrator SQA Plan.
   4. Identifying and funding an individual or group independent from the project to perform the SQA functions.
   5. Resolving and following-up on any quality issues raised by SQA.
   6. Identifying and ensuring the quality factors to be implemented in the system and software.
   7. Identifying, developing and maintaining planning documents such as the Program Management Plan, reference (h), references (e) and (f), Test Plans, and this SQA Plan.
   8. Working along all team members to fulfill project development.

c. Software Development is responsible for:
   1. Reviewing and commenting on the Orchestrator SQA Plan.
   2. Implementing the quality program in accordance with this SQA Plan.
   3. Resolving and following-up on any quality issues raised by SQA related to software design and development.
   4. Identifying, implementing, and evaluating the quality factors to be implemented in the software.
   5. Implementing the software design/development practices, processes, and procedures as defined in reference (e) and other program/project planning documents.
   6. Developing all Orchestrator module functionality including functions, classes and expected module behavior according to initial requirements.

d. Software Test is responsible for:
   1. Reviewing and commenting on the Orchestrator SQA Plan.
   2. Implementing the quality program in accordance with this SQA Plan.
   3. Resolving and following-up on any quality issues raised by SQA related to software test.
   4. Verifying the quality factors are implemented in the system, specifically software.
   5. Implementing the software test practices, processes, and procedures as defined in reference (e) and other program/project planning documents.

6. Creating Unit Test cases for the Orchestrator module.
7. Testing each function residing on the Orchestrator module.
8. Making sure all Test cases are successful when tested.

SECTION 3. RISK MANAGEMENT

.

The Orchestrator Project has developed a risk management plan as identified in Orchestrator SQA Plan. SQA will review and evaluate the technical risk analysis and any risk reduction plan. SQA reporting will confirm that the identified risks are managed in accordance with the provisions of the project's risk management plans, and that associated action items are reported, managed, and followed through to closure.

Following risks have been found on the Orchestrator Project:

- Incorrect/wrong implementation of the Orchestrator Module or other system modules such as: Databridge, Discoverer, Executor, Interpreter and Publisher modules.
- Different module functional interpretation. Each team has been assigned to work on a different system module however each team might interpret in different manner the system requirements.
- Due to the lack of time, not completing/finishing all desire features in the Orchestrator module.
- Not Successful module integration. When all modules have been completed, integration for every module will be needed. Successful integration relies on good development and corrects requirement understating.

In order to address risks, different tactics will be used during the Orchestrator Project Development, such as: Pair programming, static code reviews, unit testing, code coverage techniques and some local integration tests.

As the TBTAF Orchestrator is the brain module of the system, it will coordinate the execution and communication between the rests of the modules. Therefore static code reviews during the project development will be implemented on every iteration so bugs and failures are correctly addressed in a timely manner and fashion when discovered. Also, unit testing with Python scripts will be run against the Orchestrator module for checking its correct functionality.

SECTION 4. CHALLENGES FOR ORCHESTRATOR QA

A primary component of engineering quality into software is the conduct of technical reviews of software products, both deliverable and non-deliverable. The purpose of the technical review will be to focus on in-progress and final software. SQA will assure that technical reviews are accomplished and will selectively attend them in accordance with approved sampling techniques.

Some challenges for Orchestrator Project:

- Little technical knowledge on Python programming language.
- 2 month development and testing time for Orchestrator module.
- Complete System Integration. Every team is developing a single module from the TBTAF system, however correct implementation on the Orchestrator module will be critical as the TBTAF Orchestrator is the brain module of the system which will coordinate the execution and communication between the rests of the modules.
- General understanding on the TBTAF Discoverer, Interpreter, Executor, Publisher, Databridge and Listener modules, as the TBTAF Orchestrator module will be the brain and the coordinator between all these modules at system runtime.
- Accomplishing a successful integration between our TBTAF Orchestrator module and the other modules that resides within the TBTAF System.

The major challenge in the TBTAF Orchestrator will be the correct implementation of the API specification, where several function such as: *CreateNewProject, ParseScript, CreateTestBed, CreateTestSuite, PublishTestPlan, PublishResultReport, ExecuteTestSuite, GetTests, GetTags and module initialization* will need to be totally operating and error free as this module will be coordinating the execution and communication between the rest of the modules.

SECTION 5. ASSUMPTIONS FOR ORCHESTRATOR QA

Following assumptions could be found during the Orchestrator Project deployment:

- All teams understand the TBTAF System requirements and know how to implement/deploy their modules.
- Every module in the TBTAF System: TBTAF Discoverer, Interpreter, Executor, Publisher, Databridge, Listener and Orchestrator modules are correctly implemented.
- All modules are compatible within each other.
- Unit Test and other integration tactics have been performed against the TBTAF modules.
- Each team will successfully implement and deploy their own modules.
- Orchestrator module will be able to coordinate and communicate execution between all the TBTAF modules successfully.
- All function within the Orchestrator module will have no errors and will be totally functional.

SECTION 6. INTEGRATION TACTICS FOR ORCHESTRATOR QA

Integration is the act of bringing together smaller components into a single system that functions as one.

The TBTAF System which stands for Tag Based Test Automation Framework will include Interpreter, Discoverer, Listener, Databridge, Executor, Orchestrator subsystems within the system integration.
Integration is harder to achieve the greater the number of systems that are involved and companies often choose to have external contractors manage some or all phases of the development of the new system. In this specific System development, each team will participate with their own module integration.

Other integration challenges have to do with the lack of a coherent or unifying data structure linking all of the different systems, an unwieldy framework of multiple different applications and support systems.

Because integration is difficult to achieve all at once, a common practice is to employ a strategy of short-term, ad-hoc objectives that slowly builds towards full integration by linking various subsystems where necessary, so in this project there will be several integration rounds between all the modules.

Some integration tactics for Orchestrator Project:

- **Continuos Integration (CI).** Merging all developer working copies to a shared mainline several times a day. In this specific case, we are using *GitHub* with is a web based Git repository hosting service that offers a distributed revision control and source code management functionality to merge all code changes from every module within the TBTAF System.
- **Modular design.** By subdividing the system into smaller parts or modules that can be independently created and then used in different systems. A modular system can be characterized by functional partitioning into discrete scalable, reusable modules, rigorous use of well-defined modular interfaces, and making use of industry standards for interfaces. In this specific case the *Orchestrator module* itself.
- **Unit Testing.** Software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use (*CreateNewProject, ParseScript, CreateTestBed, CreateTestSuite, PublishTestPlan, PublishResultReport, ExecuteTestSuite, GetTests, GetTags and module initialization*).
- **Code Reviews.** By a systematic examination of computer source code. It is intended to find and fix mistakes overlooked in the initial development phase, improving both the overall quality of software and the developers' skills.
- **Pair Programming.** An agile software development technique in which two programmers work as a pair together on one workstation. In this case four team members will be conducting the pair programming during development phase.
- **Static Analysis.** To be performed as part of a Code Review (also known as white-box testing). Static Code Analysis commonly refers to the running of Static Code Analysis tools that attempt to highlight possible vulnerabilities within 'static' (non-running) source code by using techniques such as Taint Analysis and Data Flow Analysis.
- **Code Coverage (CC).** To measure used to describe the degree to which the source code of a program is tested by a particular test suite. A program with high code coverage has been more thoroughly tested and has a lower chance of containing software bugs than a program with low code coverage.

- **Integration Testing.** Will be the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before validation testing. Integration testing takes as its input modules that have been unit tested.

APPENDIX A. LIST OF ACRONYMS

AI	**Action Item**
CC	**Code Coverage**
CI	**Continuous Integration**
IEEE	**Institute of Electrical and Electronics Engineers**
SQA	**Software Quality Assurance**
TBTAF	**Tag Based Test Automation Framework**

EXTRA DOCUMENTATION

Reference to Unit testing. As part of the code design process, unit testing was implemented for the methods that were generic and useful for the Orchestrator module. They were tested by using **unittest** module from python and by creating another class called TestMethods which applied asserts to validate the expected outputs of each one of the methods against the actual output received. After using this technique we managed to:

- Make changes to the code quickly.
- Understand the design of the code that's being worked on and being able to test it right away.
- Helped the team to document the definition of the methods implemented.

Methods	Description	Number	Test Inputs	Expected Outputs	Actual Output	Test Result
isInvalidArgument		1	Argument as "Something"	FALSE	FALSE	Passed
		2	Argument as ""	TRUE	TRUE	Passed
		3	Argument as None	TRUE	TRUE	Passed
isInvalidList		1	List contains values	FALSE	FALSE	Passed
		2	List contains nothing	FALSE	FALSE	Passed
		3	List as input, with one of the values as none.	TRUE	TRUE	Passed
isInvalidPath		1	Path as ""	TRUE	TRUE	Passed
		2	Path as "Something"	TRUE	TRUE	Passed
		3	Valid Past	FALSE	FALSE	Passed
isInvalidFile		1	File Argument ""	TRUE	TRUE	Passed
		2	File Argument "Something"	TRUE	TRUE	Passed
isSupportedFilter		1	projectName is None	TRUE	TRUE	Passed
		2	projectName not in nor out nor none	FALSE	FALSE	Passed
isExistingProject		1	projectName as ""	ValueError	ValueError	Passed
		2	projectName as None	ValueError	ValueError	Passed
getServerStatusCode		1	url is ""	None	None	Passed
validateUrl		1	url is ""	ValueError	ValueError	Passed
		2	url is None	ValueError	ValueError	Passed