

Acceptance testing for the SA Risk and Vulnerability Atlas – the process

Wim Hugo, SAEON

The technical platform for the South African Risk and Vulnerability Atlas (SARVA) is currently being developed by the South African Environmental Observation Network ([SAEON](#)) on behalf of the Atlas' stakeholders, and it is important to understand the process of acceptance testing that will be followed in pursuit of a quality engineered product.

This article serves as background to an acceptance test process, and provides background information for prospective beta-testers.

SAEON follows a systems engineering approach to systems development, and in this process, test-based development plays an important part. The process that is followed is briefly discussed below.

Systems engineering process

Selection of an appropriate systems engineering approach is not simple, because the funding level of a project has a direct bearing on the amount of time and money that can be spent on the activity. It must be recognised, however, that the systems development effort results in an asset, and systems engineering has, as its main objective, to make sure that the asset meets requirements, can be safeguarded in future, and that development risks are adequately addressed.

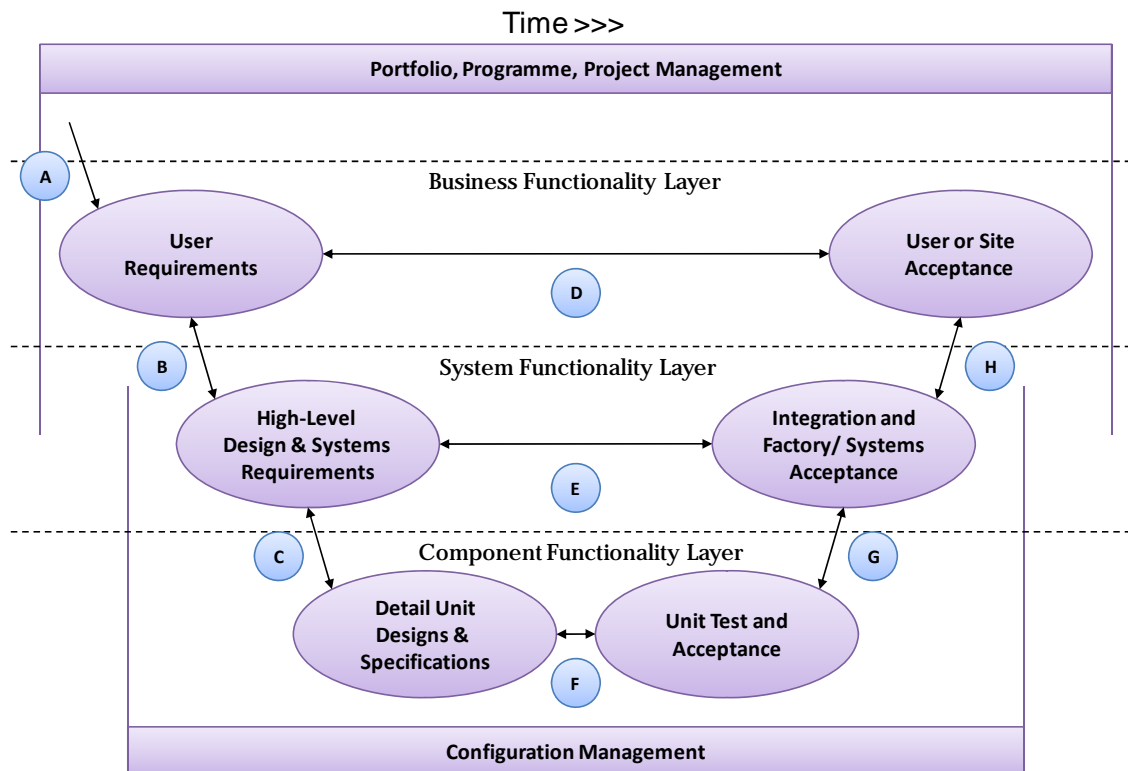
The choice of life cycle model obviously influences the type of tasks performed in the systems engineering function, and the types of documentation produced. It matters more that a life cycle model is defined, agreed between stakeholders, and followed, rather than which model is adopted.

We need to balance the systems engineering life cycle model in terms of two conflicting objectives:

1. There is an obvious requirement of minimum diligence, so that
 - a. A record of systems construction is available in sufficient detail to be transferred to a third party;
 - b. Systems developers have sufficient detail to construct a system that can meet with user requirements in an objective assessment; and
 - c. Users have sufficient detail to confirm that the documented system requirements and specifications meet with their needs.
2. On the other hand, the systems engineering function and its level of detail is strongly influenced by the size of a project and available budget.

The well-known 'V-Model' is used as a basis of our approach – not because it necessarily represents the latest thinking on systems engineering, but because it provides a coherent representation of the relationships between tasks in the process, and places the necessary emphasis on testing. It has long been a basis of systems engineering for the German Federal Government, and is shown schematically below.

Systems Engineering Framework



Salient points include:

- Three levels of detail are required for specification and testing. These reflect, broadly, the user's direct involvement, the designer's involvement, and the developer's involvement.
- Acceptance should be tested against the appropriate level of specification: This means that:
 - User acceptance is tested formally against user requirements,
 - Factory acceptance, site acceptance and system tests are performed against the systems requirements specification, and
 - Internal tests (unit tests) are performed against detailed design documentation.
- Configuration control and management need not include the user requirements set by the user or the user's acceptance tests – unless requested by the client.
- Detailed designs should map to system specifications, and systems specifications should map to user requirements specification.

One of the major management challenges for the systems engineering function is to maintain traceability between the different aspects of the life cycle - depicted in the diagram as interfaces (A), (B), (C), and so on. Commonly, in smaller projects, the lowest level of testing is not performed due to cost constraints. This means that two levels of testing are implied:

1. The Systems Engineer must ensure that the user or client can confirm a set of User Acceptance Tests (UAT). These tests typically verify that processes can be executed and are usually based on **Use Cases**.
2. The Systems Engineer must also ensure that the system meets technical specifications by way of Factory Acceptance Tests (FAT). This verifies, for example, that a meta-data standard is adhered to.

It matters very much against which environment a test is performed.

Environments

Mature systems operate in at least three distinct environments, and **tested systems are only guaranteed to work in the environment/ platform combination for which formal tests were performed**. In most instances, the three environments are:

1. Development Environment (DEV): Each contributor of systems elements, whether in-house, outsourced, or third-party, has a development environment that is under their control.
2. Likewise, each party needs to establish a formal test environment (TEST) where unit tests can be performed and which can be used as an integration test platform. This environment can physically be located at the client or at one or more developer sites. If the test site is not provided by the client, a QA site may be required to confirm integration at the client site and to perform user acceptance tests on.
3. One or more live sites (LIVE) are normally in operation that contains only the last proven integration result. Note that there may be configuration differences between different platforms/ sites.

Iterations and risk mitigation

While Agile programming methods as the basis of a life cycle model does not usually entail adequate formality from a classical systems engineering point of view, one of the advantages that it confers is the concept of small iterations and the risk mitigation that stems from this.

Systems development risk can be mitigated in two generic ways by manipulation of the scope of work:

- By limiting the scope of an iteration, and making releases more frequently. In the case of the Risk Atlas, this iteration frequency can be as short as 2-3 months. Each iteration is a complete systems life cycle.
- By making use of widely used and proven components (open source supports this objective), which removes the component scope of work from the release.

Releases

For each iteration, the following releases can apply:

- A **Null** release. This release implements interfaces and nothing more – in essence it allows simple integration tests that prove integration infrastructure and interface implementations without (necessarily) any back-end functionality.
- Internal development is done up to the point of an **Alpha** release. This release has the following characteristics:
 - It is never deployed beyond the DEV environment.
 - It substantially implements the requirements specification set for it.
 - It has limited error handling.
 - It has not been formally tested by the client.
- Test releases are usually the same as a **Beta** release. This implies that the release:

- Meets requirement scope and can be released formally,
- Implements all required ancillary requirements (such as error handling, security, and other non-business functionality).
- Provides that the known issues, which should have been addressed in the release, can be satisfactorily signed off.
- This release is always made available on the **TEST** environment.
- A formal release is only made once beta tests have been concluded and issues, if any, have been addressed.

The table below describes the relationship between generic environments and generic releases:

(Minor) Release	DEV	TEST	LIVE
Null	Yes	Yes	Never
Alpha	Yes	No	Never
Beta	Yes	Yes	Site Acceptance Tests
Formal Release	Yes	Yes	Yes

Tests and Acceptance

Types of Test and Purpose

User Acceptance Tests (UAT)	User Acceptance Tests (UAT) prove that the system meets the stated user requirements.
Factory Acceptance Tests (FAT)	Factory Acceptance Tests (FAT) prove that the system adheres to its specifications and designs.
Integration Tests (INT)	Integration Tests prove that the combined parts of an application or system can function together correctly. This means that individual software modules are combined and tested as a group. In many cases, integration can be proven against components that simulate the behaviour of external components rather than the components themselves. It is a special type of FAT.
Unstructured Tests	While unstructured tests can undoubtedly lead to issue identification, it is important to agree that the results have no formal status (See below). Beta testing often leads to this type of test.
Negative Testing	Unless specifically requested or defined, negative testing is not performed.
Load Testing (LT)	These have to be formally specified, with desired results/ benchmarks clearly stated.

Requirements for Formal Test Status

To be considered a formal test. The following criteria need to be met:

Test Schedule or Script	Formal tests require a test script or schedule that details the following: <ul style="list-style-type: none">• A traceable reference to the URS or specification item that requires the test.• Instructions for each test to be performed.• Expected result• Actual Result.
System Under Test	The system under test needs to be clearly defined. This entails the following: <ul style="list-style-type: none">• Details of the Iteration and Release under test.• Details of the platform being used.• Details of the test data set being used.
Test Data Sets	It is vitally important that tests be conducted against a special test data set. There are three major reasons for this: <ul style="list-style-type: none">• It must be possible to achieve the results expected in the test schedule using data examples corresponding to the test data set; on more than one occasion.• It must be possible to repeat tests for different releases using the same

	test data. • It also eliminates test failures due to data issues.
Logging Test Results	Test results need to be classified in respect of two aspects (see below). These are: • Error severity (impact on system operation) • Root Cause
Test Operators/ Teams	Tests are always performed by designated roles. Tests performed outside of these roles may, as with unstructured tests, sometimes add value, but they do not have any status.
User Feedback and Issue Resolution	Users identify issues and suggestions for improvement under normal system use. These issues and suggestions need to be resolved through a managed process that adds value. Prerequisites include: • Users are properly trained (i.e. understand expected system behaviour) • Users are registered (legitimate) account holders.

Who performs formal tests?

- Development Team (DT) – Vendor/ Service Provider personnel.
- Test Team (TT) – Vendor personnel.
- Technical Acceptance Team (TAT) – Test Team, Technical Client Representative.
- User Test Team (UTT) – Test Team, Client Representative, One or more users.

(Minor) Release	DEV	TEST	LIVE
Null	DT	TT	Never
Alpha	DT	TT	Never
Beta	TT	TAT	TAT
Formal Release	TT	UTT	UTT

Issue resolution

Systems development success is, to a large extent, dependent on efficient issue resolution.

To achieve efficiency, the following guidelines are important:

- Issues need to be contextualized in terms of its formality. There is a large difference between an issue raised by a test team under test conditions, and an informal issue raised by a user under non-test conditions. Specifically, the contractual implications are very different in terms of the responsibilities of the service provider or systems developer.

- It also needs to be contextualized in terms of severity. Again, there is a big difference between an issue (raised in a live environment) that prevents users from executing system functions and an issue raised by testers in a non-live environment.
- Furthermore, the root cause of the issue is of vital importance. Depending on the systems engineering life cycle model in use, the details of the root cause terminology may differ, but the central point remains: issues are often not due to programming error, but to faulty data, mis-specification of requirements, inappropriate use of the system, and so on. A comprehensive list, based on the proposed life cycle model, is provided below.
- Finally, issues must be assigned to future releases and iterations on the basis of severity. This is a critical component of prioritized mitigation and improvement of systems.

An issue management system or environment that is visible to all participants can assist with the efficiency of resolution, but is not a prerequisite.

Severity classification

Priority	Description
Severity 1	System level failures involving the loss of an operational/ critical business function or feature/ capability. Includes both system and customer impacting faults. Any highly critical system or service outage that results in severe degradation of overall performance.
Severity 2	Part of critical business function/ feature. Error is not entirely preventing its operation. Includes both system and customer impacting faults. Any major degradation of system or service performance that impacts significantly, impairs system/ operator control, or operational effectiveness. A suitable and agreed work-around must exist.
Severity 3	Incidental fault or feature. Non-service impacting. Most "new" requirements. Can be addressed in the next major release.

Error root cause classification

It is recommended that all errors are reported using a standardised attempt to define the root cause.

Item #	Specification Description	Discussion
1	No Issue or Not Repeatable	The issue cannot be reproduced in the specified environment.
2	Scope Extension - Requirement not documented	The requirement was expressed by the users, but isn't known (documented in an issue list, expressed in a formal document)
3	BRS is incorrect	Business Requirement/ Impact Assessment does not capture the need correctly.
4	URS is incorrect	User Requirement Specification does not capture the need correctly.
5	SRS is incorrect	System Specifications do not capture the needs correctly.
6	BRS misinterpreted (in translating to URS)	URS incorrectly translated from BRS.
7	URS misinterpreted (in translating to SRS)	SRS incorrectly translated from URS.

8	SRS misinterpreted (in translating to implementation).	Implementation does not reflect SRS.
9	Programming Error.	Programming bug or error.
10	Test Script is incorrect	Test does not accurately test the feature.
11	Test Data is incorrect or incomplete	Test data in use
12	Configuration Error	Incorrect combination of components and/ or test data
13	Error in Live Data	Error occurred in live data, but cannot be replicated with test data.
14	Platform Error	Error occurs on a specific platform, but not on the test platform.
15	Unstructured Test: Error Identified	Negative test or an error occurred outside of the formal test step.
16	Unstructured Test: Improvement Suggested	Not a test step, but a suggestion for improvements or change.