Best practices on writing great Test cases

On best practices

This document provides some guidelines for addressing the specification of test cases during testing.

Here you may find some recommended tips to perform better testing in general and also whenever using Xray.

Although named "best practices", please consider this document informative and not binding, since not every aspect is covered in it and proper evaluation needs to be performed to ensure your needs are addressed. As Xray evolves and testing methodologies also evolve, these guidelines and your process may need to be adjusted and evolve likewise.

Overview

Testing is crucial for the products that you build and that your customers use.

In the same way that you demand high-quality code from an architectural and strict development point-of-view, your test cases should also have the same high-quality.

As you struggle to make releases more often, while maintaining quality as a priority, you struggle with limited resources and limited time.

Thus, you need to make choices in order to perform the right testing with the time that you have, while maintaining a certain level of confidence.

In this document you can find some guidelines for enabling great testing, which, as we'll see, is not limited to writing test cases.

Enabling great testing

Testing quality does not just depend on the executed Tests and bugs found during that process. Testing is a lot more.

But to have great testing, you'll need first of all to have a testable SUT, which is properly described and detailed in clear, well-defined requirements, and reviewed from the start by great testers whose skills helped build a better product through awesome testing.

A testable SUT

In order for a system to be testable, it must provide certain characteristics (i.e. it must be built in such a way that promoves them) including:

- **observability**: test results must be easily observable
- **controlability**: ability to control the state of the SUT
- **automatability**: provides mechanisms that allow automation to be implemented
- **stability & availability**: must be available and stable enough so testing can be performed efficiently in a reliable manner
- **simplicity & comprehension**: must be understandable, so that testing can be performed

Clear, concise and correct requirements

In order to specify great Tests for existing requirements, first of all they need also to meet certain quality criteria, by incorporating some key characteristics:

- **consistency**: internally and externally; by using consistent terminology and by making sure requirements are non-contradictory internally and between each other
- **complete and concise, yet short**: requirement scope must be well defined and limited, so that it can be implemented correctly; some caution should be taken in order to avoid, hard to digest descriptions
- **correct**: addresses correctly the business/stakeholders needs
- **unambiguous**: clear, not subject to different interpretations
- **quantitative** (not qualitative): quantitative aspects can be measured, while qualitative ones are ambiguous
- **verifiable**: a requirement must provide ways so that it can be verified; having acceptance criteria is a common practice to address this

Being a great tester

To perform valuable testing, a tester needs many skills. Some of these include the ability to:

- know the SUT, its components and the communication/interactions between them as much as possible
- understand the use cases
- understand the value proposition (what does the SUT try to solve for the end users/customers)
- have some level of understanding of the internal architecture of the SUT
- be part of the team; work together with other team members, including developers, BA, etc
- understand what is critical for the customer (using feedback mechanisms, such as metrics, support among other)

A great Test

What makes a Test, a great Test?

There are a set of characteristics that all scripted tests should have. The following ones are just a few of them:

- **goal oriented**: focused on one goal, one purpose, gives meaning/value to the Test itself
- **simple & short**: simple Tests are easier to understand and to execute
- **consistent**: in the terminology being used, non-contradictory
- **reliable & deterministic**: how consistent test results are between different runs, executed under the same conditions
- **objective:** how clear the test is so that different testers achieve the same results
- **reviewed**: by having the feedback of others, your Test will become better
- **valid**: to what extent will test results address the purpose of the test, accordingly with the acceptance criteria defined for the requirement
- **helpful on RCA (root cause analysis)**: helps not just on identifying a problem but on understanding the root cause
- **maintainable**: how easy is it to update changes in the related requirement
- **discoverable**: a Test should be easy to find, by multiple criteria, whenever you need it
- **fast**: a Test should be executed as fast as possible as a means to reduce effort; this is specially crucial for automated testing where lengthy Tests can burden the CI/CD process
- **independent**: Tests that are self-contained are easier to manage
- **parallel**: can be run on parallel; this is specially important for automated testing in order to reduce the overall execution time

Whenever performing other types of testing, such as exploratory testing, some of the former do not need to be considered due to the nature of the practice, including objectivity and consistency among other.

Tests in Xray

Xray provides three different "Test Types:"

- **Manual**: a scripted test composed of steps; mainly used for manual testing but it can also be used as an abstraction of an automated test
- **Cucumber**: a cucumber Scenario/Scenario Outline, normally executed in automated context (although it can also be executed manually)
- **Generic**: an abstraction of some Test that is neither Manual nor Cucumber; normally, corresponds to an automated test implemented in some framework (e.g. JUnit, TestNG, Robot, custom framework). It can also be used as a means to abstract an exploratory test/session

However, one consequence of the above is that there is no explicit way to identify whether a Test is automated or not. However, you may define your own rules for that.

Best Practices

Process

Testers are part of the team

Involve testers whenever reviewing and discussing requirements or user stories with the development team; make testers part of the team and shift your testing to the left side of your development life cycle.

Requirements/stories should be clear and have acceptance criteria

Clear requirements are essential to have the correct understanding of what aims to be addressed by the requirement and the related business needs.

Make sure they are reviewed; you can use Jira's workflow **status** in the related issue for this.

Test using the whole Pyramid

Have tests for the different layers of the Test Pyramid, starting with unit tests up to E2E/GUI tests.

To remember:

- Don't forget unit tests; your code should be covered by them
- Focus more on integration and service tests, where the logic/core features of the SUT can be evaluated
- GUI tests can be simpler to make, yet easier to break and harder to maintain; however, they're also needed

Manage changes on requirements

Changes on requirements/stories should be reviewed by testers.

- Acceptance criteria may need to be also updated accordingly
- Related Tests may need to be reviewed
  - You may transition the Tests workflow **status** to some specific status to signal that; a post-function on a workflow transition of the requirement can help with that

Shift Left Testing

Review the requirements with the "testing team", write Tests and run them as soon as possible. By making testing part of the whole development life-cyle, requirements will be more clear, understandable and testable.

Perform exploratory testing

Define test objectives, goals, and areas of opportunity that allow the tester to use her own creativity and skill.

Check here how to perform Exploratory Testing using Xray in a very basic way.

Requirements/stories should be clear and have acceptance criteria

Clear requirements are essential to have the correct understanding of what aims to be addressed by the requirement and the related business needs.

Make sure they are reviewed; you can use Jira's workflow **status** in the related issue for this.

Store Tests alongside your other project development related issues

Although Xray is quite flexible and supports many different project organization scenarios, we recommend you to have your testing related artifacts together with other project related ones, such as requirements/user stories, bugs, tasks, etc. This approach provides a self-contained project, it's easier to understand and manage and, best of all, promotes team collaboration.

Organize Tests properly from the start

Tests must be organized in proper ways, either by using lists (i.e. Test Sets) or folders (i.e. within the project's Test Repository).

But besides this more structured way, Tests can and should be properly "organized" right form the start, independently of how they are to be grouped later on.

This can be achieved by "tagging" the Tests adequately and by identifying the reason for the Test to exist in the first place; with this you can:

- Assign them **labels**, **component(s)** or other fields
- Link them to requirements/stories; if you don't have requirements, consider creating some so you can track your testing results from the requirements/features perspective
- Use a specific field to classify the type of Test (e.g. integration, UI)
- Use a specific field to classify the nature of the Test (i.e. manual, automated)

It's important that your process clearly states how to perform the earlier points, namely the tagging. Otherwise you will end up with similar Tests but tagged in different ways.

Specification

Tests with a purpose

The reason for a Test to exist in the first place should be its purpose/goal.

- Make use of the **Summary** and **Description** fields to define the purpose of the Test

All the necessary and clear steps

Clear, non-ambiguous steps (actions and expected results) avoid different interpretations and thus different results.

Tests should have the necessary steps to validate their purpose:

- If you have very few steps, possibly you're assuming too much and you need to detail some further "implicit"/missing steps
- If you have many steps, probably you're validating too many things at the same time; it's best to have tailored test cases for each scenario that you wish to validate; try to have separation of concerns

Mark the Tests so they can easily be found/managed

Use **labels** or **specific custom fields**; if using labels, keep in mind that Jira's default label type of fields is case-sensitive and does not provide a way to limit its values; a more restricted custom field type may be more appropriate.

Tests specific for a given Environment

If Tests are specific for a given environment, use the Environment field to clearly identify that.

Whenever scheduling these Tests for execution, make sure to use the Test Environments feature by setting the Test Environments field on the Test Execution.

More info on Test Environments here.

blocked URLAvoid too many dependencies

A Test that depends on other Tests being executed previously, even by a certain order, makes the Test harder to understand and manage.

Try to avoid dependencies in the first place. If really needed, try to isolate them as much as possible on Pre-Conditions.

Note that a Pre-Condition has only an open text field that is used to describe it; it does not provide semantics to explicit mentioning dependencies to other Test cases.

Missing Precondition/requisite for running a Test

If a certain condition is necessary to run a Test, it's best to abstract in a Pre-Condition.

This makes sense only if you foresee it as something reusable, that is useful for other Tests.

Relative importance of Tests for a given requirement

Use **Priority** field to distinguish between different Test cases; Priority is standard field from Jira and thus should be the preferred field for this purpose.

Avoid UI/visual dependencies

Don't make Tests dependent on very specific UI aspects, such as the position or on text labels, unless they're UI/visually aimed.

Test in different configuration scenarios

Perform testing using the same test procedure but in different conditions, such as different combinations of enabled features.

In this case, the preferred approach would be the following one:

- create multiple clones of the same Test
- assign each Test to a specific Pre-Condition, where each one corresponds to a different combination of enabled features

An alternate approach would be using Test Environments and define each one as a specific feature combination. However, this does not scale well if you have several features which would lead to the creation of many Test Environments. Also, you may prefer to use Test Environments as a means to identify the target environment instead (i.e. browser, mobile device).

Few Tests (or few testing) per requirement

A requirement covered by few test cases may be a symptom of lacking test cases, testing only the successful path.

- Create more Tests using white and black box techniques

Dozens of Tests per requirement

A requirement covered by large dozens of test cases may be a symptom that the requirement is too complex or vague.

- Decompose the requirement in more simple, more testable requirements

Avoid having very few Tests, that only validate the obvious

Sometimes testers look at a requirement and write one or two Tests that mimic exactly the description of the requirement, which can be quite simplistic and give a totally wrong sense of confidence

- Have in mind the acceptance criteria and write several test cases for each of them
    - Use techniques such as Boundary Value Analysis (BVA), Equivalence Class Partitioning, Decision Table based testing among other ones, to improve coverage without requiring tons of testing effort

Exercise Equivalence Class Partitioning and BVA

As a means to provide greater coverage, Equivalence Class Partitioning and Boundary Value Analysis can provide enhanced coverage without growing the Test suite too much.

- Clone Tests and use the "data" column to put the values for the inputs of each partition/class
- If doing Equivalence Class Partitioning, don't specify exact values for the class, to have some randomness; if later on you find a bug, you may create a Test case with the exact value to verify the fix

Few Tests (or few testing) per requirement

A requirement covered by few test cases may be a symptom of lacking test cases, testing only the successful path.

- Create more Tests using white and black-box testing related techniques

Promove reusability and avoid cloning Tests specifications

If Tests are exactly the same, in theory there should be no need to have clones.

Whenever talking about clones, we may be talking about explicit clones (i.e. cloning the whole Test issue, with all its steps) or implicit/embedded clones (i.e. where the steps of an existing Test are used in another more high-level Test case).

In general, users may clone Tests for different purposes:

- Perform the same Test but against different data
    - Use test parameterization, when it's available; until then, there are different "workaround"/interim approaches that can be followed, where the first is probably the best one:
        1. Clone Test and use the "data" column to put the values for the different parameters. This is valid if you don't need to filter later on by these parameters, otherwise the parameters can be persisted as specific custom fields on the Test issue, and thus easily be queriable or reported against. Some caution should be taken with the later, in order to avoid huge amounts of custom fields which will burden management and also affect performance of Jira instance. This approach allows to track the coverage per each different data values combination.

2. Specify parameters at the moment of execution. Since a Test Execution can only contain a Test once (i.e. it can only contain 1 Test Run of a given Test case), this would mean creating as many Test Executions as the possible combinations of input values. Parameters and their values could be defined as custom fields on the Test Execution; care should also be taken in order to avoid huge amounts of custom fields

- Cover/test different requirements, of different projects; in this case projects may be somehow similar and may represent slight variations /conditions of the "main requirement"
    - You may use the same Test for this (probably the Test would reside in some common project); however, if you do so, then the Test result will affect related requirements in the same way (unless you change the Separation of Concerns setting). Instead, and because you're probably really talking about different requirements, the best is to have specifically designed Tests for each individual requirement
- E2E Test composed of steps, where these steps are essentially existing Test cases
    - In this case, the best is to design the E2E Test in such a way that its steps are more simple and validate only what is the purpose of the E2E Test itself (e.g. instead of having an E2E Test with all the steps of a login related Test case, the E2E Test can have just one step where the user is expected to login using given username/password)
- Tests having some common initial steps
    - Abstract those common initial steps in reusable Pre-Conditions linked with all those Tests

Performance considerations

1. Avoid scripted Tests with dozens of steps
2. Avoid Tests linked to many different requirements; try to have just one covered requirement per Test
3. Avoid huge amounts of Tests per requirement; consider more grained requirements

Quick Checklist

Use this to quickly evaluate if you're on the right track or not. This list is a very condensed sum up of many practices mentioned earlier.

#	Description	Why it matters?
1	Have you defined a process for your testing, with guidance covering the specification?	Having a well defined process ensures that users follow the same procedures and use tools in similar ways. Thus, similar things will be done in similar ways which allow better collaboration grounded on proper understanding. This is key to effectively working as a team and avoid problems later on.
2	Is the Test understandable? Can you understand the scope and the actions you need to perform to perform the testing?	If the Test is ambiguous, then its results will also be likewise. An understandable Test is mostly a useless and invalid test for the purpose it seeks to address.
3	Are your Tests being reviewed?	Having reviews on the Tests makes them more clear and robust.
4	Can you clearly identify the type of Test? Are specific fields/labels being used for this purpose?	Properly identifying tests by multiple criteria will allow you to find them whenever needed later on, for regression or sanity testing for example.
5	Do you have a way to clearly identify deprecated Tests?	As your Test suite/database grows, some Tests will become useless and you only have them for tracking purposes. It's crucial to have a way to clearly identify these Tests and exclude them from coverage calculations.
6	Do your Tests have a medium amount of steps?	Few or too much steps are signs that something is either missing/assumed or that too much is being done in the scope of the Test. Tests with a medium amount of step tend to be clearer, are more focused and easier to manage.
7	Have you materialized all assumptions in Pre-Conditions?	Using Pre-Conditions fosters reusability by making these assumptions visible and manageable.
8	Are you performing testing at the different layers /levels?	It is really important to have unit tests. But it also important to have tests at different levels of the well-known Test Pyramid, because each level has its own value and addresses different concerns.
9	Are Tests covering one and just one requirement?	Tests linked to requirements allow traceability and evince their goal. If a Test just covers one requirement, then it means that its purpose is focused and the results impact will be more easy to interpret.
10	Are you just testing the happy path?	Edge cases are many times the source of problems. But besides it, by understanding the context of the Test, of the requirement and its implementation, additional Tests can be designed to cover potential impacted areas of the SUT.
11	Are you just performing manual testing?	You should complement manual testing with automated testing and exploratory testing. Each practice has its own benefits and complements each other.