2.5 Test case and Test scenario

Test case is a concept that provides detailed information on what to test, steps to be taken and expected result at the same time while Test Scenario provides one-line information about what to test.

When is documentation of Test Cases important?

- The client has asked for the same as part of the project.
- There is no time constraint
- Testers are unexperienced or not familiar with the product.
- Company policy

To meet deadlines and making the process easy and transparent, the way of documenting can be changed. Documentation of test scenarios could be an acceptable approach when time is the constrained and the only expectation is foolproof testing. You could save the time on documentation and utilize it for testing.

Test Suite vs **Test Scenario**:

- Making a payment (test scenario)
- Making a payment using a credit card (test suite)
- Making a payment using a voucher (test suite)
- Making a payment using X type of payment method (test suite)
- Testing using valid data (test case)
- Testing using invalid data (test case)
- Testing using a card with no funds (test case)
- Testing using an expired voucher (test case) and so on…

Test Scenarios examples: **HOMEWORK!**

High-Level Scenario: Login Functionality

Low-Level Scenarios:

- To check Application is Launching
- To check text contents on the login page
- To check Username field
- To check Password field
- To check Login Button and cancel button functionality

2.5.1 How to write Test Case?

Writing test cases brings some sort of standardization. A test case can have the following elements:

1. **Test Suite Id** - The ID of the test suite to which this automated test case belongs
2. **Test Case Id** — The ID of Test Case

3. **Title/Subject** — Use a Strong Title
4. **Precondition and Assumptions test**
5. **Description** — Steps to be executed
6. **Test data** —The test data, or links to the test data, that are to be used while conducting the test
7. **Expected result** — The expected result of the test
8. **Actual result** — The actual result of the test or test case execution
9. **Status** — Passed/Failed (optional Not Executed (Pending, Dropped, Blocked), Passed with Exception, etc.)
10. **Related requirement** — The ID of the related requirement this test case relates/traces to.
11. **Created by**
12. **Date of creation**
13. **Executed by**
14. **Date of execution**
15. **Test Environment** — The environment (Hardware/ Software/ Network) in which the test was executed
16. **Comments**

Characteristic of a good test case:

- **Accurate**: Exacts the purpose
- **Economical**: No unnecessary steps or words
- **Traceable**: Capable of being traced to requirements.
- **Repeatable**: Can be used to perform the test over and over
- **Reusable**: Can be reused if necessary

2.5.2 Tips for Writing Tests

Important factors Involved in Writing Process:

a. TCs are prone to regular revision and update
b. TCs are prone to distribution among the testers who will execute these
c. TCs are prone to Clustering and Batching
d. TCs have a tendency of inter-dependence
e. TCs are prone to distribution among the developers

Tips to Write Effective Tests:

a. Keep it simple but not too simple; make it complex but not too complex
b. After documenting the Test cases, review once as Tester
c. Bound as well as ease the Testers

d. Be a Contributor
e. Never Forget the End Use

Most Common Problems in Test Cases:

a. Composite steps
b. Application behavior is taken as expected behavior
c. Multiple conditions in one case

Useful Tips and Tricks for writing tests:

a. Is your Test Document in Good Shape
b. Do not forget to cover the Negative Cases
c. Have Atomic Test Steps
d. Prioritize the Tests
e. Sequence Matters
f. Add Timestamp and the Tester Name to the Comments
g. Include Browser/OS Details
h. Keep two separate sheets — 'Bugs' & 'Summary' in the Document

2.5.3 Examples of test scenarios

Functional Test Scenario: Upload document as a proof of payment

Business requirement: On the web page, users can upload a document as proof of payment.

User Story: As a User, I want to be able to upload the document as proof of payment.

Acceptance criteria:

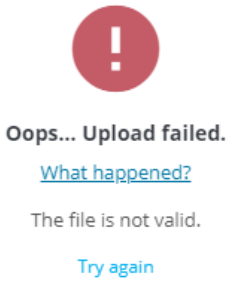
1. Allowed file format .PDF, .DOC, .DOCX, .PPT, .PPTX, JPG, JPEG, .GIF, .PNG, .MSG.
2. Allowed maximum size 25MB.
3. Error message for not supported format for uploaded file displayed:

   "*Wrong file format. Supported formats are: .PDF, DOC, .DOCX, .PPT, .PPTX, JPG, JPEG, .GIF, PNG, .MSG.*"
4. Error message for an oversize uploaded file displayed:

   "*File that you uploaded is too large. The maximum allowed size is 25MB.*"
5. If proof of payment successfully uploaded users get an appropriate message.
6. **Visual design:**

7. If upload failed users get an appropriate message.

   Visual design:



Test case examples

1. The test is written “step by step”:

   Test Scenario: Verify that uploaded file format for proof of payment is allowed on the web page

   Test Steps:

   1) Logged in on the web page: https//exampes.eg
   2) Click on Upload button for proof of payment
   3) A new popup is displayed
   4) Select a File with a type (Test data)
   5) Click OK button
   6) Proper message is displayed

   Expected Result: File should be uploaded /not uploaded (Positive/Negative).

   Actual Result: File uploaded /not uploaded, as expected.

2. The test is written in so-called *Gherkin language*

   Summary: Verify that uploaded file format for proof of payment is allowed on web page

   Scenario:

   Given logged in on the web page: https//exampes.eg
   And click on Upload button for proof of payment
   And new popup is displayed
   When select file with "<format>"
   And click on OK button
   Then verify the "<message>" is shown

   Examples:

format	message
.GIF	Upload complete
.PDF	Upload complete
.DOC	Upload complete

.PPT	Upload complete
.JPG	Upload complete
.PNG	Upload complete
.TXT	Wrong file format. Supported formats are: .XLS, .PDF, .DOC, .PPT, JPG, .TXT.

Non-functional Test Scenario: The icon for uploading status per design

Test Scenario: Verify proper design uploaded file format for proof of payment

Test Steps:

1. Logged in on the web page: https//examples.eg
2. Click on Upload button for proof of payment
3. A new popup is displayed
4. Select a File
5. The icon for uploading status is displayed per design

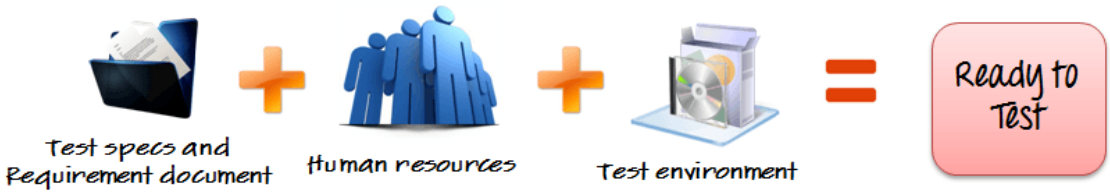
Expected Result: The icon for uploading status should be displayed per design.

Actual Result: The icon for uploading status is displayed per design.

Attachment: Visual design for successful upload and Visual design for unsuccessful upload

LINK: Functional Testing Vs Non-Functional Testing: What's the Difference?

2.5.4 Test Plan (resources: **LINK**)



So far, we have covered Test Case and Test Scenario. **On a bigger scale, we have Test Plan**. It is a detailed document that describes the approach we will take in testing efforts.

It consists of:

- analyzing the product,
- designing the test strategy,
- defining test objectives,
- defining testing criteria,
- planning out resources needed,
- planning test environment,
- scheduling and estimation,
- determining test deliverables.

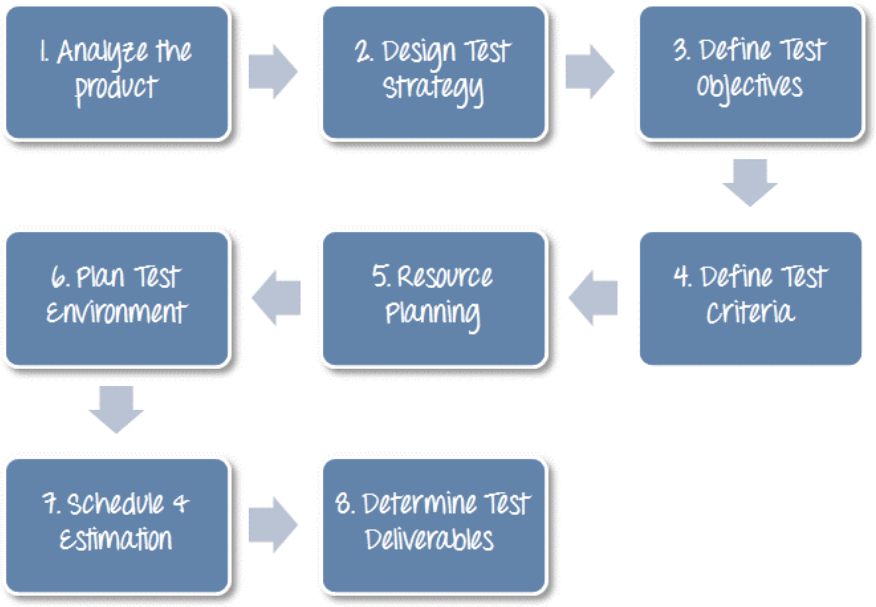


Figure 13. Test plan flow. https://www.guru99.com/what-everybody-ought-to-know-about-test-planing.html

Most of the managers are adamant on making test plans.

So why should you do it?

Test plan helps in communicating test objectives and estimations how long should each step take and how much effort is needed, as well as the risks. If Test Plan is made, then it is easier for everyone else, developers, managers, customers, to understand the details of testing.

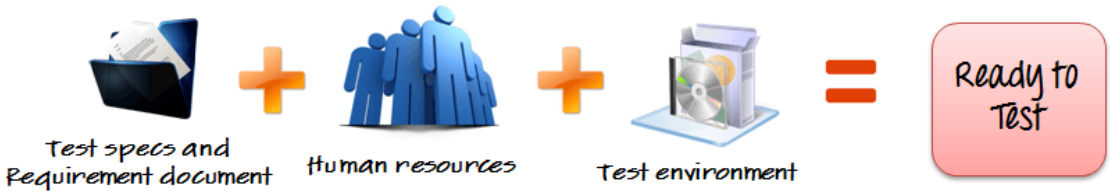
From the testers perspective, it is a guideline as to what to focus on, how to plan time and schedule. You may look at it as a tool to help you, as a tester, understand the process, potential downfalls, and ease of organizing. Furthermore, it helps you quickly explain the strong and weak sides of testing efforts. It also underlines the responsibilities of individuals during testing, who does what and when.

Test plan differs in different development models which will be explained in a later chapter, and here, we will cover the agile model, as currently the most used model of development. Unlike the waterfall model, in an agile model, a test plan is written and updated for every release. The agile test plan includes types of testing done in that iteration like test data requirements, infrastructure, test environments, and test results.

Typical test plans in agile includes:

- Testing Scope
- New functionalities which are being tested
- Level or Types of testing based on the feature's complexity
- Load and Performance Testing
- Infrastructure Consideration

- Mitigation or Risk Plan
- Resourcing (**Who** will test? **When** will the test occur?)
- Deliverables and Milestones



2.6 Defect Management

2.6.1 What is a Bug

Bug is a defect which is introduced by a person's error (mistake). Error can be made in requirement elicitation, can be introduced while writing software code. If the error in code is executed, that it can lead to failure, but may not if it occurs in some very specific conditions which are rarely or never executed.

The term "bug" was used in an account by computer pioneer Grace Hopper, who publicized the cause of a malfunction in an early electromechanical computer. A typical version of the story is:

In 1946, when Hopper was released from active duty, she joined the Harvard Faculty at the Computation Laboratory where she continued her work on the Mark II and Mark III. Operators traced an error in the Mark II to a moth trapped in a relay, coining the term bug. This bug was carefully removed and taped to the log book. Stemming from the first bug, today we call errors or glitches in a program a bug.

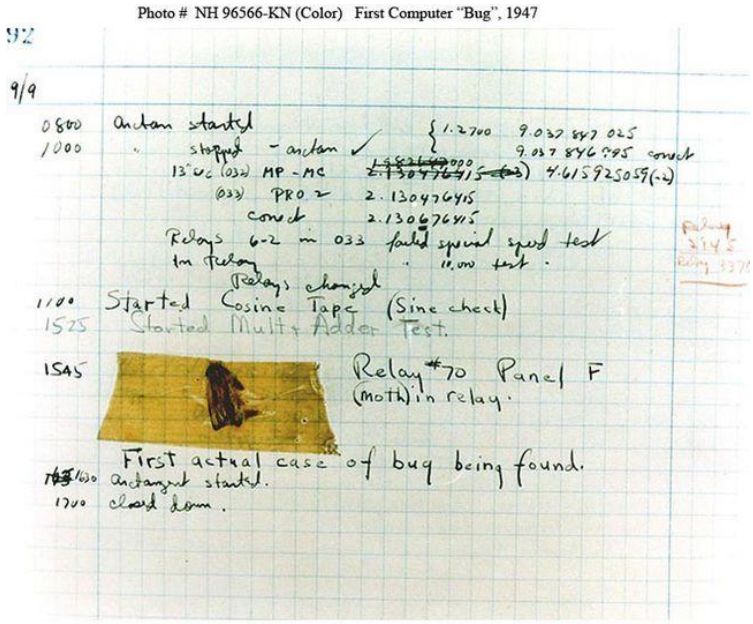


Figure 16: First reported case of "bug"

Defect Life Cycle or **Bug Life Cycle** is the specific set of states that a Bug goes through *from discovery to defect fixation*.

2.6.2 Bug Life Cycle Status

The number of states that a defect goes through varies from project to project. Below lifecycle diagram, covers all possible states:

- **New**: When a new defect is logged and posted for the first time. It is assigned a status as NEW.
- **Assigned**: Once the bug is posted by the tester, the lead of the tester approves the bug and assigns the bug to the developer team.
- **Open**: The developer starts analyzing and works on the defect fix.
- **Fixed**: When a developer makes a necessary code change and verifies the change, he or she can make bug status as "Fixed."
- **Pending retest**: Once the defect is fixed the developer gives a code for retesting the code to the tester. Since the software testing remains pending from the testers end, the status assigned is "pending request."
- **Retest**: Tester does the retesting of the code at this stage to check whether the defect is fixed by the developer or not and changes the status to "Re-test."
- **Verified**: The tester re-tests the bug after it got fixed by the developer. If there is no bug detected in the software, then the bug is fixed, and the status assigned is "verified."
- **Reopen**: If the bug persists even after the developer has fixed the bug, the tester changes the status to "reopened". Once again, the bug goes through the life cycle.
- **Closed**: If the bug is no longer exists then tester assigns the status "Closed."
- **Duplicate**: If the defect is repeated twice or the defect corresponds to the same concept of the bug, the status is changed to “duplicate.”
- **Rejected**: If the developer feels the defect is not a genuine defect then it changes the defect to "rejected."
- **Deferred**: If the present bug is not of a prime priority and if it is expected to get fixed in the next release, then status "Deferred" is assigned to such bugs
- **Not a bug**: if it does not affect the functionality of the application then the status assigned to a bug is "Not a bug’.

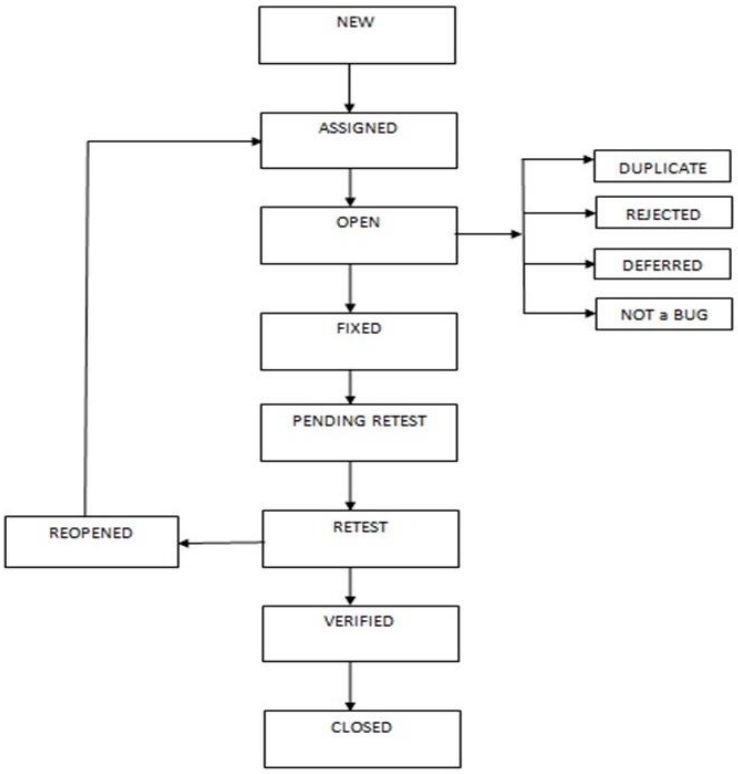


Figure 17: defect lifecycle

2.6.3 Bug Priority & Severity

Bug priority indicates the importance or urgency of fixing a defect. Though priority may be initially set by the Software Tester, it is usually finalized by the Project/Product Manager.

Priority can be categorized into the following levels:

- Urgent
- High
- Medium
- Low

At a high level, priority is determined by considering the following:

- Business needs for fixing the defect
- Defect Severity / Impact
- Defect Probability / Visibility
- Available Resources (Developers to fix and Testers to verify the fixes)
- Available Time (Time for fixing, verifying the fixes and performing regression tests after the verification of the fixes)

Defect Priority needs to be managed carefully in order to avoid product instability, especially when there is a large number of defects.

Defect Severity determines the defect's effect on the application.

Severity can be categorized into the following levels:

- High (Severe)
- Major
- Minor
- Low

In the bug report, Severity and Priority are normally filled in by the person writing the bug report but should be reviewed by the whole team.

High Severity - High Priority bug

This is when the major path through the application is broken, for example, on an eCommerce website, every customer gets an error message on the booking form and cannot place orders, or the product page throws an Error 500 response.

High Severity - Low Priority bug

This happens when the bug causes major problems, but it only happens in very rare conditions or situations, for example, customers who use very old browsers cannot continue with their purchase of a product. Because the number of customers with very old browsers is very low, it is not a high priority to fix the issue.

High Priority - Low Severity bug

This could happen when, for example, the logo or name of the company is not displayed on the website. It is important to fix the issue as soon as possible, although it may not cause a lot of damage.

Low Priority - Low Severity bug

For cases where the bug doesn’t cause disaster and only affects a very small number of customers, both Severity and Priority are assigned low, for example, the privacy policy page takes a long time to load. Not many people view the privacy policy page and slow loading doesn’t affect the customers much.

The above are just examples. It is the team that should decide the Severity and Priority for each bug.

Examples:

1. *High Severity — High Priority*: Application for booking not keeping a reservation.
2. *High Severity — Low Priority*: The user has an old version of Browser which doesn’t let him make a reservation.
3. *Low Severity — High Priority*: The logo of a company is the wrong color.
4. *Low Severity — Low Priority*: Privacy Policy loading slowly.

2.6.4 Bug template

Finding a bug is one thing, but documenting it is just as important.

- **Summary**: Explain what defect you found, on which browser, environment and project
- **Priority**
- **Severity**
- **Description**: Steps to reproduce, every step click by click how to reproduce the defect
- **Expected result:** What supposed to happen?
- **Actual result:** the observed failure
- **Attachment:** Make a screenshot or video as a proof that defect exists
- **Device:** What type of hardware are you using? Which specific model?
- **OS:** Which version number of the OS has displayed the issue?
- **Testing App Version:** Date, time, version and build of the software

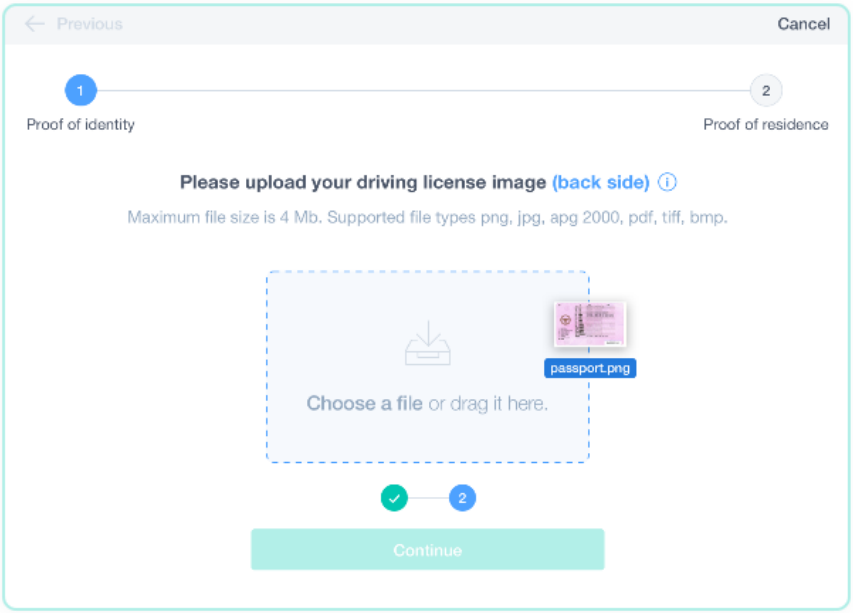
Example 1:

Observed issue: The user uploaded the document with no size as a proof of payment, the document is empty.

Summary: Empty document can be uploaded as proof of payment — *Functional defect*

Priority: High (if proof of payment is a mandatory document), Low (if proof of payment is optional)

Severity: Low



Description:

Environment: Pre-prod

Browser: Google Chrome Version 78.0.3904.70

Steps to reproduce:

1. Logged in on the web page: https//exampes.eg
2. Click on Upload button for proof of payment
3. A new popup is displayed
4. Select a File with a size OKB (Test data)
5. Click OK button

Expected result: File shouldn't be uploaded. Validation message should be presented: "In order to continue with request, you need to upload proof of payment."

Actual result: File is added as proof of payment. No validation message is presented.

Attachment: Video_attachment_bug.mp4

Example 2:

You are testing purchasing application, and you have selected your native currency in which you want your prices to be displayed. You encounter a bug reported like this:

Summary	wrong currency
Expected result:	use the right currency (euros)
Actual result:	wrong currency is used (pounds)

Are you aware of what is wrong with the application? Does it always show prices in pounds? Should it always show prices in euros? If you reported this bug, initially it would make sense, because you know which specific functionality you were testing, and hence to what this actually refers to. But imagine you come back to this bug few months later and see this description. Would it make sense to you? Or if someone else reported this? *This report doesn't have nearly enough information.* Although you should keep reported bugs as simple as possible and clean, you should still explain in detail what the use case was, and what functions incorrectly.

Summary	[Product list for specific search] Mixed currencies for the products.
Expected result:	As the location and currency are selected in the header (France-EUR), all product prices should presented in Euros.
Actual result:	On the product list, on the first page, at least 5 products have a different currency (pounds) than the currency selected as default, in which other products are displayed (euros).

The description is still short, and clean, but, as you can see, it gives much more detail. *This way you save time for your colleagues trying to figure out what is wrong in the first place.* Once you come back to this bug, you will yourself know, right of the bat, what was wrong then.