1 Fundamentals of Testing

1.1 What is Testing?

Software systems are an integral part of life, from business applications (e.g., banking) to consumer products (e.g., cars). Most people have had an experience with software that did not work as expected. Software that does not work correctly can lead to many problems, including loss of money, time, or business reputation, and even injury or death. Software testing is a way to assess the quality of the software and to reduce the risk of software failure in operation.

A common misperception of testing is that it only consists of running tests, i.e., executing the software and checking the results. As described in section 1.4, software testing is a process which includes many different activities; test execution (including checking of results) is only one of these activities. The test process also includes activities such as test planning, analyzing, designing, and implementing tests, reporting test progress and results, and evaluating the quality of a test object.

Some testing does involve the execution of the component or system being tested; such testing is called dynamic testing. Other testing does not involve the execution of the component or system being tested; such testing is called static testing. So, testing also includes reviewing work products such as requirements, user stories, and source code. Another common misperception of testing is that it focuses entirely on verification of requirements, user stories, or other specifications. While testing does involve checking whether the system meets specified requirements, it also involves validation, which is checking whether the system will meet user and other stakeholder needs in its operational environment(s).

Test activities are organized and carried out differently in different lifecycles (see section 2.1).

1.1.1 Typical Objectives of Testing

For any given project, the objectives of testing may include:

- To prevent defects by evaluate work products such as requirements, user stories, design, and code
- To verify whether all specified requirements have been fulfilled
- To check whether the test object is complete and validate if it works as the users and other stakeholders expect
- To build confidence in the level of quality of the test object
- To find defects and failures thus reduce the level of risk of inadequate software quality
- To provide sufficient information to stakeholders to allow them to make informed decisions, especially regarding the level of quality of the test object
- To comply with contractual, legal, or regulatory requirements or standards, and/or to verify the test object's compliance with such requirements or standards

The objectives of testing can vary, depending upon the context of the component or system being tested, the test level, and the software development lifecycle model. These differences may include, for example:

- During component testing, one objective may be to find as many failures as possible so that the underlying defects are identified and fixed early. Another objective may be to increase code coverage of the component tests.
- During acceptance testing, one objective may be to confirm that the system works as expected and satisfies requirements. Another objective of this testing may be to give information to stakeholders about the risk of releasing the system at a given time.

1.1.2 Testing and Debugging

Testing and debugging are different. Executing tests can show failures that are caused by defects in the software. Debugging is the development activity that finds, analyzes, and fixes such defects. Subsequent

confirmation testing checks whether the fixes resolved the defects. In some cases, testers are responsible for the initial test and the final confirmation test, while developers do the debugging, associated component and component integration testing (continues integration). However, in Agile development and in some other software development lifecycles, testers may be involved in debugging and component testing.

ISO standard (ISO/IEC/IEEE 29119-1) has further information about software testing concepts.

1.2 Why is Testing Necessary?

Rigorous testing of components and systems, and their associated documentation, can help reduce the risk of failures occurring during operation. When defects are detected, and subsequently fixed, this contributes to the quality of the components or systems. In addition, software testing may also be required to meet contractual or legal requirements or industry-specific standards.

1.2.1 Testing's Contributions to Success

Throughout the history of computing, it is quite common for software and systems to be delivered into operation and, due to the presence of defects, to subsequently cause failures or otherwise not meet the stakeholders' needs. However, using appropriate test techniques can reduce the frequency of such problematic deliveries, when those techniques are applied with the appropriate level of test expertise, in the appropriate test levels, and at the appropriate points in the software development lifecycle. Examples include:

- Having testers involved in requirements reviews or user story refinement could detect defects in these work products. The identification and removal of requirements defects reduces the risk of incorrect or untestable features being developed.
- Having testers work closely with system designers while the system is being designed can increase each party's understanding of the design and how to test it. This increased understanding can reduce the risk of fundamental design defects and enable tests to be identified at an early stage.
- Having testers work closely with developers while the code is under development can increase each party's understanding of the code and how to test it. This increased understanding can reduce the risk of defects within the code and the tests.
- Having testers verify and validate the software prior to release can detect failures that might otherwise have been missed, and support the process of removing the defects that caused the failures (i.e., debugging). This increases the likelihood that the software meets stakeholder needs and satisfies requirements.

In addition to these examples, the achievement of defined test objectives (see section 1.1.1) contributes to overall software development and maintenance success.

1.2.2 Quality Assurance and Testing

While people often use the phrase quality assurance (or just QA) to refer to testing, quality assurance and testing are not the same, but they are related. A larger concept, quality management, ties them together. Quality management includes all activities that direct and control an organization with regard to quality. Among other activities, quality management includes both quality assurance and quality control. Quality assurance is typically focused on adherence to proper processes, in order to provide confidence that the appropriate levels of quality will be achieved. When processes are carried out properly, the work products created by those processes are generally of higher quality, which contributes to defect prevention. In addition, the use of root cause analysis to detect and remove the causes of defects, along with the proper application of the findings of retrospective meetings to improve processes, are important for effective quality assurance.

Quality control involves various activities, including test activities, that support the achievement of appropriate levels of quality. Test activities are part of the overall software development or maintenance process.

Since quality assurance is concerned with the proper execution of the entire process, quality assurance supports proper testing. As described in sections 1.1.1 and 1.2.1, testing contributes to the achievement of quality in a variety of ways.

1.2.3 Errors, Defects, and Failures

A person can make an error (mistake), which can lead to the introduction of a defect (fault or bug) in the software code or in some other related work product. An error that leads to the introduction of a defect in one work product can trigger an error that leads to the introduction of a defect in a related work product. For example, a requirements elicitation error can lead to a requirements defect, which then results in a programming error that leads to a defect in the code.

If a defect in the code is executed, this may cause a failure, but not necessarily in all circumstances. For example, some defects require very specific inputs or preconditions to trigger a failure, which may occur rarely or never.

Errors may occur for many reasons, such as: * Time pressure * Human fallibility * Inexperienced or insufficiently skilled project participants * Miscommunication between project participants, including miscommunication about requirements and design * Complexity of the code, design, architecture, the underlying problem to be solved, and/or the technologies used * Misunderstandings about intra-system and inter-system interfaces, especially when such intrasystem and inter-system interactions are large in number * New, unfamiliar technologies

In addition to failures caused due to defects in the code, failures can also be caused by environmental conditions. For example, radiation, electromagnetic fields, and pollution can cause defects in firmware or influence the execution of software by changing hardware conditions.

Not all unexpected test results are failures. False positives may occur due to errors in the way tests were executed, or due to defects in the test data, the test environment, or other testware, or for other reasons. The inverse situation can also occur, where similar errors or defects lead to false negatives. False negatives are tests that do not detect defects that they should have detected; false positives are reported as defects, but aren't actually defects.

1.2.4 Defects, Root Causes and Effects

The root causes of defects are the earliest actions or conditions that contributed to creating the defects. Defects can be analyzed to identify their root causes, so as to reduce the occurrence of similar defects in the future. By focusing on the most significant root causes, root cause analysis can lead to process improvements that prevent a significant number of future defects from being introduced.

For example, suppose incorrect interest payments, due to a single line of incorrect code, result in customer complaints. The defective code was written for a user story which was ambiguous, due to the product owner's misunderstanding of how to calculate interest. If a large percentage of defects exist in interest calculations, and these defects have their root cause in similar misunderstandings, the product owners could be trained in the topic of interest calculations to reduce such defects in the future.

In this example, the customer complaints are effects. The incorrect interest payments are failures. The improper calculation in the code is a defect, and it resulted from the original defect, the ambiguity in the user story. The root cause of the original defect was a lack of knowledge on the part of the product owner, which resulted in the product owner making an error while writing the user story. The process of root cause analysis is discussed in ISTQB-CTEL-TM and ISTQB-CTEL-ITP.

1.3 Seven Testing Principles

A number of testing principles have been suggested over the past 50 years and offer general guidelines common for all testing.

1. **Testing shows the presence of defects, not their absence**
   Testing can show that defects are present, but cannot prove that there are no defects. Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found, testing is not a proof of correctness.
2. **Exhaustive testing is impossible**
   Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases. Rather than attempting to test exhaustively, risk analysis, test techniques, and priorities should be used to focus test efforts.
3. **Early testing saves time and money**
   To find defects early, both static and dynamic test activities should be started as early as possible in the software development lifecycle. Early testing is sometimes referred to as shift left. Testing early in the software development lifecycle helps reduce or eliminate costly changes (see section 3.1).
4. **Defects cluster together**
   A small number of modules usually contains most of the defects discovered during pre-release testing, or is responsible for most of the operational failures. Predicted defect clusters, and the actual observed defect clusters in test or operation, are an important input into a risk analysis used to focus the test effort (as mentioned in principle 2).
5. **Beware of the pesticide paradox**
   If the same tests are repeated over and over again, eventually these tests no longer find any new defects. To detect new defects, existing tests and test data may need changing, and new tests may need to be written. (Tests are no longer effective at finding defects, just as pesticides are no longer effective at killing insects after a while.) In some cases, such as automated regression testing, the pesticide paradox has a beneficial outcome, which is the relatively low number of regression defects.
6. **Testing is context dependent**
   Testing is done differently in different contexts. For example, safety-critical industrial control software is tested differently from an e-commerce mobile app. As another example, testing in an Agile project is done differently than testing in a sequential software development lifecycle project (see section 2.1).
7. **Absence-of-errors is a fallacy**
   Some organizations expect that testers can run all possible tests and find all possible defects, but principles 2 and 1, respectively, tell us that this is impossible. Further, it is a fallacy (i.e., a mistaken belief) to expect that just finding and fixing a large number of defects will ensure the success of a system. For example, thoroughly testing all specified requirements and fixing all defects found could still produce a system that is difficult to use, that does not fulfill the users' needs and expectations, or that is inferior compared to other competing systems.
   See Myers 2011, Kaner 2002, Weinberg 2008, and Beizer 1990 for examples of these and other testing principles.

1.4 Test Process

There is no one universal software test process, but there are common sets of test activities without which testing will be less likely to achieve its established objectives. These sets of test activities are a test process. The proper, specific software test process in any given situation depends on many factors. Which test activities are involved in this test process, how these activities are implemented, and when these activities occur may be discussed in an organization's test strategy.

2 Markdown Showcase

Paragraphs are separated by a blank line.

2nd paragraph. *Italic*, **bold**, and `monospace`. Itemized lists look like:

- this one
- that one
- the other one

> Block quotes are written like so.
> They can span multiple paragraphs, if you like.

Use 3 dashes for an em-dash. Use 2 dashes for ranges (ex., "it's all in chapters 12–14"). Three dots ... will be converted to an ellipsis (only if you enable the `smartEllipses` option).

2.1 Subchapter

Here's a numbered list (use `hashEnumerators` option if you want to use hashes):

1. first item
2. second item
3. third item

Here's a code sample:

```
# Let me re-iterate ...
for i in 1 .. 10 { do-something(i) }
```

As you probably guessed, indented 4 spaces.

Or use fenced code (with `markdown` v2.4):

```
# Let me re-iterate ...
for i in 1 .. 10 { do-something(i) }
```

If you have `minted` loaded in your project you get syntax-highlighted code:

```
<?php
    print("Hello World");
?>
```

2.1.1 Sub sub section

Now a nested list:

1. First, get these ingredients:
    - carrots
    - celery
    - lentils
2. Boil some water.

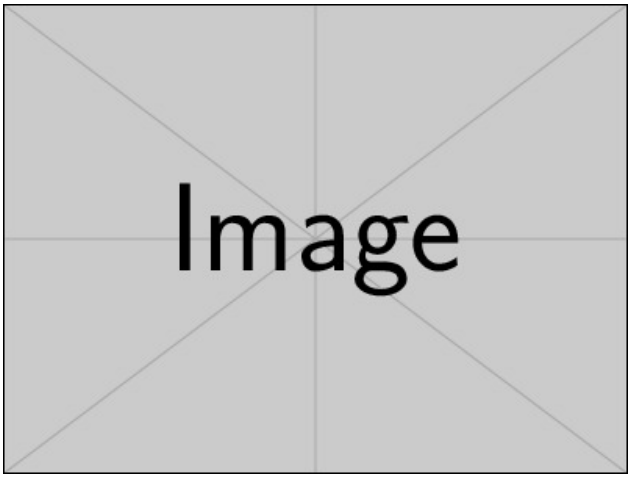

Figure 1: An exemplary image

3. Dump everything in the pot and follow this algorithm:

```
find wooden spoon
uncover pot
stir
cover pot
balance wooden spoon precariously on pot handle
wait 10 minutes
goto first step (or shut off burner when done)
```

   Do not bump wooden spoon or it will fall.

Here's a link to a website[1]. And now[2] you can also use inlined footnotes with `inlineFootnotes`.

A horizontal rule follows.

Here's a definition list (with `definitionLists` option):

apples Good for making applesauce.
oranges Citrus!
tomatoes There's no "e" in tomatoe.

and images can be specified like so, and cross-referencing works if you add a `fig:` to the label: **Figure 1**

If you enable the `hybrid` option, You can mix LaTeX code in Markdown! Inline math equations go in like so: $\omega = d\phi/dt$. Displaymath too:

$$I = \int \rho R^2 dV \tag{1}$$

And note that you can backslash-escape any punctuation characters which you wish to be displayed literally, ex.: 'foo', *bar*, etc.

Citations are now supported with `markdown` v2.4; but beware of underscores in BibTeX keys (best avoided)! When they work, they look like [**?**] or see [**?**, p.26].

[1] `<http://foo.bar>`
[2] with `markdown` v2.4

Right	Left	Default	Center
12	12	12	12
123	123	123	123
1	1	1	1

Table 1: Demonstration of pipe table syntax.

As of `markdown` v2.8.0 you can use PHP's pipe table syntax, if you load the `pipeTables` option. You don't have to line up the pipes exactly; somehow it works out. If you also add the `tableCaptions` option, you can add a caption too! Note that there must be an empty line after the caption.