# Week 2

## Introduction to Software Testing

We shall discuss following topics in this week's notes:

- What is software Testing?
- What are the objectives of performing testing? That means what we try to achieve?
- What are the different levels of Testing? - mainly we talk about Unit, integration, system, and manual testing.
- Testing Pyramids and Cones.
- Principles of Testing
- Manual and Automated Testing
- Some definitions

### 1. Introduction

In the previous chapter, we introduced the notion of software quality - in short, meeting explicit and implied customer needs. As professional software engineers, we should aim to reliably deliver software with confidence the delivered product meets its quality goals. In this week's notes, we shall focus on Software Testing and will not get back to the quality aspects with respect to the formal and executable artefact until Week 10.

### 2. What is Software testing?

Some people define software testing (for simplicity, we will just use "testing" from here on in) as including all software quality assurance activities. However, that is not standard. For the purposes of this unit, we will take (as a starting point) the IEEE SWEBOK definition of testing:

> *"Software testing consists of the dynamic verification that a program provides expected behaviours on a finite set of test cases, suitably selected from the usually infinite execution domain."*

*"Dynamic"* means that the program is actually executed, rather than static techniques. *"Expected behaviour"* means that we have some method by which we can decide whether the behaviour matches the requirements. Given that we have a finite amount of time and either a very large or infinite number of potential inputs (and, as SWEBOK notes, "*inputs*" describe the entire state of the environment where that affects the program), we have to select a finite number of inputs to conduct testing with.

This is not a perfect definition, but a reasonable one. In practice, much testing is not conducted on a complete software system, but some component of it. Furthermore, in some cases our

knowledge of what the expected behaviour is, is limited. Consider a scientific simulation system to determine the behaviour of a model of a physical environment (for instance, a metal alloy), one that is difficult or impossible to observe directly. How can we know if the code is buggy or not given that we do not actually know what the expected behaviour is?

This definition also means that formal methods and inspections are not testing, though they are most definitely software quality assurance activities. Within this definition, there are a range of ways testing activities can be classified, and we will look at a few. The first one we will look at is classifying testing by the type of quality property we are trying to assess. Not all of these quality properties will be covered in detail, but you need to at least be aware that they exist, and that they can be tested.

## 3. Objectives (Goals) of Software Testing

Software Testing has different goals or objectives. The major objectives of software testing are as follows.

### 3.1 Functional Correctness

This tends to be what people think of when "*testing*" is mentioned without qualification. Does the software produce the outputs consistent with the software specification, given the inputs?

Faults, failures, and other mistaken terminology: when a piece of software does not behave consistently with the specification, this is termed a failure. The underlying cause of this failure is called the fault.

The connection between faults and failures is, in practice, more complicated than that. Many failures can be prevented from occurring again with many different changes to the code. To give a simple example, consider a method that returns the sum of a list that crashes on an empty list. You could fix it by modifying the method to return 0 when the input is an empty list. You could also fix it by ensuring that no empty lists are passed as an argument to the method. The term "bug" is an informal one; it is generally used to mean fault that occurs in code but can also be used to mean failure.

In any case, as we already hinted at in week 1, there is a very important limit to what testing can achieve, as first expressed by Dijkstra back in 1969:

*"Testing shows the presence, not the absence of bugs."*

This observation leads to the notion of "*debug testing*"; testing directed at revealing as many bugs in the software under test as possible given the available testing resources. This might sound appealing, and it is the basis for many common testing methods. It's not, however, automatically the best way to deliver the most reliable software, if the bugs found are unlikely to occur in actual use. An alternative approach is "operational testing", where the tests are designed to reflect actual usage.

We will concentrate mainly on how to test for functional correctness and will present techniques for doing so in the next section.

### 3.2 User Acceptance

User acceptance testing, as the name implies, is where the system as a whole is tested to ensure that it meets users' high-level business requirements. In the "*verification and validation*" spectrum, this is very much at that validation end. For custom software written for a client, this is often conducted by that client or client representative.

It's often that case, particularly in systems developed using a waterfall approach, that problems in requirements elicitation finally make themselves obvious in user acceptance testing, as earlier SQA efforts based on flawed requirements won't reveal problems.

While it is a lot better to identify problems in user acceptance testing than in actual use, it's much better if they are found in earlier phases of software quality assurance, not least system testing.

### 3.3 Performance

You can also conduct testing to ensure that the software meets resource utilization requirements. While time - whether clock or CPU time - is the performance requirement that first comes to mind, it is by no means the only one. In some applications, disk or memory usage may be important; in mobile applications the amount of energy used to perform tasks may be key.

Performance testing is a complex and subtle task; it is tightly coupled to specifying resource utilization requirements in sensible ways. In a safety-critical system, for instance, it may be the case that there are hard clock-time requirements for actions to occur. For instance, consider an anti-lock braking system for a car; the requirements might specify that brake lockups must always be detected within a certain time interval. In web applications you might not have "hard" response time requirements. However, predicting usage and testing appropriately to ensure that your system can cope with the predicted usage, often requires statistical modelling to accompany performance testing.

Performance testing, particularly for web and database applications, is often automated with specialized tools.

### 3.4 Security

> *"Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break."* -- Bruce Schneier

In theory, security is just another functional correctness property. In practice, security is one of the hardest things to provide as a developer, and hardest to assess from a QA perspective - except in the negative, as it is often distressingly easy to show systems are insecure. The

difference is that in the case of security issues, any failure, no matter how obscure, difficult to trigger, or trivial, will be exploited to the maximum extent possible. The unlikeliness of a security flaw is no defence against an attacker who can exercise your system in precisely the way required to trigger the issue.

Even more than other types of functional correctness, you cannot QA security into already written code at the end of the development process - it must be an integral part of the process, including requirements gathering and risk assessment. Even when security QA is done right, perfect security is impossible. One of the hard-earned lessons of both physical and IT security is that keeping out very skilled and well-resourced attackers is extremely difficult. Addressing security also often results in high costs, and systems that are compromised in other aspects such as usability.

Specialized security testing comes in several forms, on top of standard functional correctness testing. "*Fuzz testing*" involves bombarding the system under test with random, or randomly modified data. While this can be used for general correctness testing, it is found some high-profile security flaws, including the infamous "*Heartbleed*" security bug in 2014.

Network applications are built using a wide variety of software infrastructure, such as databases, web servers, "frameworks", and security libraries such as OpenSSL. Bugs are regularly found and patched in these, but for many reasons applications continue to use vulnerable versions of infrastructure underneath. Vulnerability scanners systematically test applications against these known bugs.

Penetration testing uses tools such as these, as well as the ingenuity of security professionals, who will systematically search for ways to perform unauthorised actions on systems to demonstrate that their security is not as specified. In some cases, penetration testing will examine not only the software, but also the organizational context in which it is deployed. For instance, testing whether users will supply their passwords to a purported "*system administrator*" who calls them.

### 3.5 Usability

Usability testing is the technique to evaluate a product by testing it on users. Usability testing is inherently a manual process, involving getting users to attempt to use the system and collecting feedback on how they interact with it. Capturing that feedback can be done in a variety of ways, from simply asking the users, to recording their behaviour with software tools, and even cameras and microphones in certain circumstances.

A challenging part of usability testing may be finding appropriately representative users, particularly for applications with a large and diverse user base. An interface which is fine for adult English-speaking programmers may not be suitable for toddlers, residents of an aged care facility, or medical professionals in an emergency room, to take a few examples!

Large-scale usability testing is often the domain of usability specialists, many of whom have backgrounds in psychology or related disciplines. If usability is a particularly high priority for your system, engaging their expertise may be wise.

Usability testing is another area which we will not cover in any detail in the unit but is extremely important for many applications. Ultimately, software exists to enable people to achieve their goals.

### 3.6 Reliability

Reliability testing is designed to assess how reliable the software under test is. That is, how regularly will it fail to perform as specified. This is a different goal to testing for functional correctness, where we hope to expose areas where it does not perform to specification so that they can be fixed.

To perform reliability testing in a sane way, you need a mathematical model of how the software is going to be used - an operational profile - so that the testing can reflect that. In practice, few software systems have the effort put into them to collect good operational profiles, but without them, a reliability estimate is just a guess.

### 3.7 Robustness

Reliability testing determines how often things break. Robustness testing, by contrast, looks at the consequences of breakage. That is, failures are injected into the system (for instance, by shutting down part of a system in the middle of an operation) and the behaviour of the system is monitored to ensure that the consequences of the failures are acceptable.

### 3.8 Regression

Regression testing involves repeating some testing of a system to ensure that software behaviour has not changed (except in some desirable way) after modification. Regression testing can be carried out to evaluate any of the above properties.

Automated regression testing is standard practice in modern development, and automated test tools make it relatively easy to do. We will discuss regression testing in more detail later.

## 4. Test Levels

A software test might be targeted at anything from an individual method, to the entire system under development. In some cases, it might even involve testing the interaction between the system under development with its environment. For example, a device controlled by embedded software being tested as a whole, or a network utility tested to ensure that it is compatible with pre-existing network devices.

In IEEE-speak, a *'test level'* refers to the scope of the piece of software being tested. Another way to phrase it is the "test target". A software system goes through four stages of testing before it is actually deployed. These four stages are known as unit, integration, system, and acceptance

level testing. The first three levels of testing are performed by a number of different stakeholders (and are part of SWEBOK Standard) in the development organization, whereas acceptance testing is performed by the customers. The four stages of testing have been illustrated in the form of what is called the classical V model as shown in figure 1. We shall unit, integration, system in this unit.

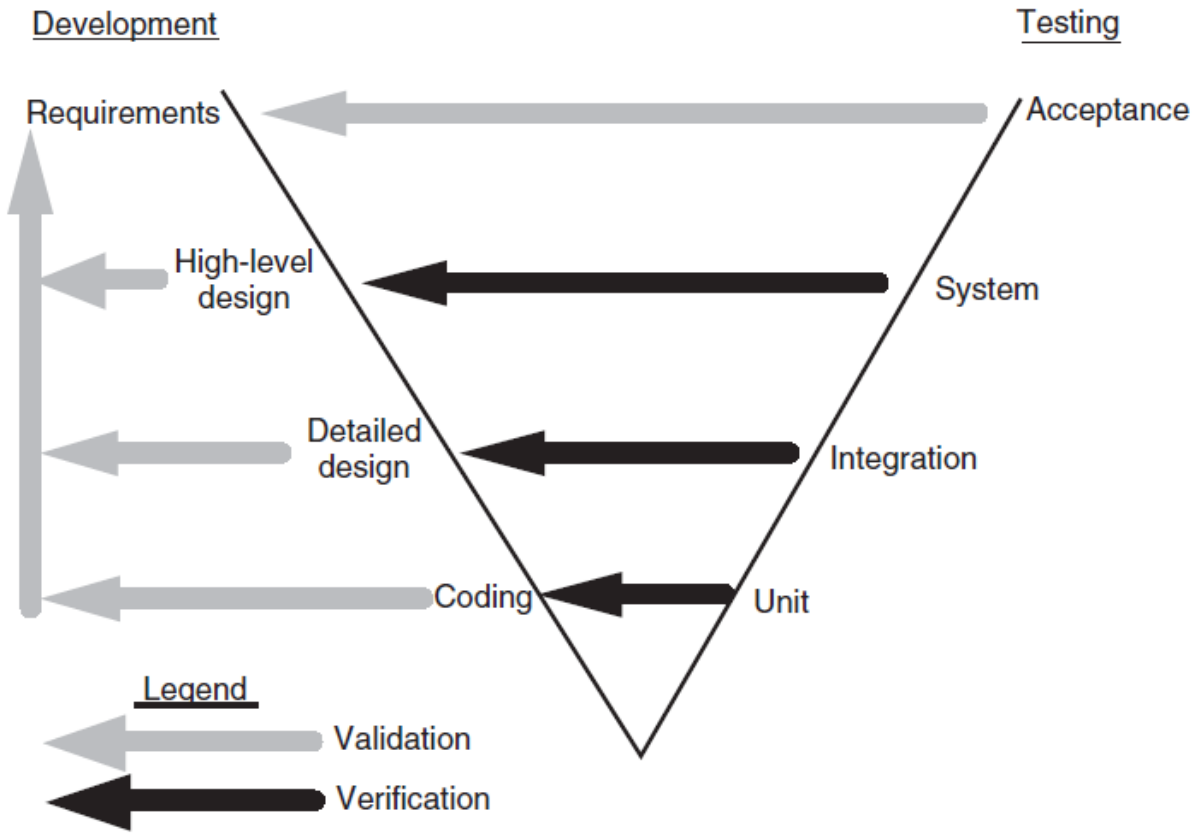


*Figure 1: Development and Testing Phases in V model.*

### 4.1 Unit Testing

Unit testing involves the testing, in isolation, of the smallest separately testable components of a larger software system. In the context of modern programming environments, a unit is generally a class or module. Unit testing is typically done by the programming team, and possibly the individual programmer who wrote the code.

In some situations, our goal is indeed to test a single feature of the software, "purposefully ignoring the other units of the systems". Like we have done so far. When we test units in isolation, we are doing what we call unit testing.

Defining what a 'unit' is, is challenging and highly dependent on the context. A unit can be just one method or can consist of multiple classes. According to Roy Osherove's unit testing definition: "A unit test is an automated piece of code that invokes a unit of work in the system. And a unit of work can span a single method, a whole class or multiple classes working together to achieve one single logical purpose that can be verified."

As with any testing strategy, unit testing has advantages and disadvantages:

### Advantages

- Speed of execution. A unit test usually takes just a couple of milliseconds to execute. Fast tests give us the ability to test huge portions of the system in a small amount of time.
- Unit tests are easy to control. We have a high degree of control on how the unit tests exercise the system. A unit test tests the software by giving certain parameters to a method and then comparing the return value of a method to the expected result. The parameters and expected result values are very easy to be adapted/modified in the test.
- Unit tests are easy to write. The PyUnit (and JUnit for Java Language) test framework is a very nice and easy framework to write unit tests in. It does not require a lot of setup or additional work to write the tests.

Of course, unit testing also has some disadvantages:

### Disadvantages

- One of which is the reality of the unit tests. A software system is rarely composed of a single class or fulfils a single logical purpose. The large number of classes in a system and the interaction between these classes can make the system behave differently in its real application than in the unit tests. Hence, the unit test does not perfectly represent the real execution of a software system.
- Another disadvantage that follows from this is that some bugs simply cannot be caught by means of unit tests. They happen only in the integration of the different components (which we are not exercising in a pure unit test).

We will go through the framework of Automation Testing (Unit Testing in Python using PyUnit) in detail from Week 7 onwards.

## 4.2 Integration Testing

Integration testing involves testing interactions between components within a system. It is generally assumed that integration testing is conducted after unit testing is completed and does not reveal a failure.

Early testing (and software engineering) textbooks devoted reams of paper to explain the merits of different integration strategies; first you'd combine modules with others that you immediately interacted with, then you'd test those groups of modules together … and the order in which you did this was considered of huge significance. There were "top-down", "bottom-up", and various other strategies, and lots of maths explaining the amount of "stub" and "driver" code you would need to write to fake the code that the group of modules you were testing interacted with.

This kind of approach might have been necessary when "the system" did not exist until very late in the piece. However, modern development projects nearly always use continuous integration to manage the process of integration. This changes how integration testing works. In short, as soon as possible into the project, *"the system"* exists, and is compile-able and runnable, even if it cannot do very much. New functionality gets integrated continuously into this skeleton as soon as possible, and integration tests are added to the system's automated test suite. This minimizes the amount of "stub" and "driver" code required, and more importantly tends to shake out integration faults much more quickly than a stepwise integration process that only gives you a running system at the very end.

Let us look at a real example. When a system has an external component, e.g., a database, developers often create a class whose only responsibility is to interact with this external component. Now, instead of testing all the system's components, we just want to test this class and its interaction with the component it is made for. One class, one (often external) component. This is integration testing.

In integration testing, we test multiple components of a system, but not the whole system altogether. The tests focus on the interaction between these few components of the system.

The advantage of this approach is that, while not fully isolated, devising tests just for a specific integration is easier than devising tests for all the components together. Because of that, the effort of writing such tests is a bit higher than when compared to unit tests (and lower when compared to system tests). Setting up the external component, e.g., the database, to the state the tester wants, requires effort.

Can we fully replace system testing with integration testing? After all, it is more real than unit tests and less expensive than system tests. Well… No. Some bugs are really tricky and might only happen in specific situations where multiple components are working together. There is no silver bullet.

We will discuss continuous integration frameworks extensively in the unit in Week 8

### 4.3 System Testing

This involves testing the behaviour of an entire system. It should be noted that while acceptance testing is conducted on an entire system, not all system testing is acceptance testing.

Unit tests indeed do not exercise the system in a realistic way. To get a more realistic view of the software, we should run it in its entirely. All the components, databases, front end, etc, running together. And then devise tests.

When executing it, we are doing what we call system testing. In practice, instead of testing small parts of the system in isolation, system tests execute the system as a whole. Note that we

can also call system testing as black-box testing, because the system is some sort of black box to the tests. We do not care or actually know what goes on inside of the system (the black box) as long as we get the expected output for a given input.

#### Advantages

- The obvious advantage of system testing is how realistic the tests are. The system executes as if it is being used in a normal setting. This is good. After all, the more realistic the tests are, the higher the chance of it actually working will be.
- The system tests also capture the user's perspective more than unit tests do.
- Faults that the system tests find, can then be fixed before the users would notice the failures that are caused by these faults.

#### Disadvantages

However, system testing also has its downsides.

- System tests are often slow when compared to unit tests. Having to start and run the whole system, with all its components, takes time.
- System tests are also harder to write. Some of the components, e.g. databases, require complex setup before they can be used in a testing scenario. This takes additional code that is needed just for automating the tests.
- Lastly, system tests tend to become flaky. Flaky tests mean that the tests might pass one time but fail the other time.

System testing is automatable in some circumstances but is not always because the programming and execution environment often does not make it easy to do so. Where system testing is conducted manually, the task is often handled by testing specialists, partly because software engineers are far too expensive to be wasted on it. If it is conducted manually, the tests must be documented carefully, a process we will spend some time in weeks 3 and 4.

### 4.4 Acceptance Testing

After the completion of system-level testing, the product is delivered to the customer. The customer performs their own series of tests, commonly known as acceptance testing. The objective of acceptance testing is to measure the quality of the product, rather than searching for the defects, which is objective of system testing.

## 5. The Testing Pyramid and Cone

We discussed four levels of tests: unit, integration, system. How much should we do of each level? A famous diagram that helps us in discussion is the so-called testing pyramid.

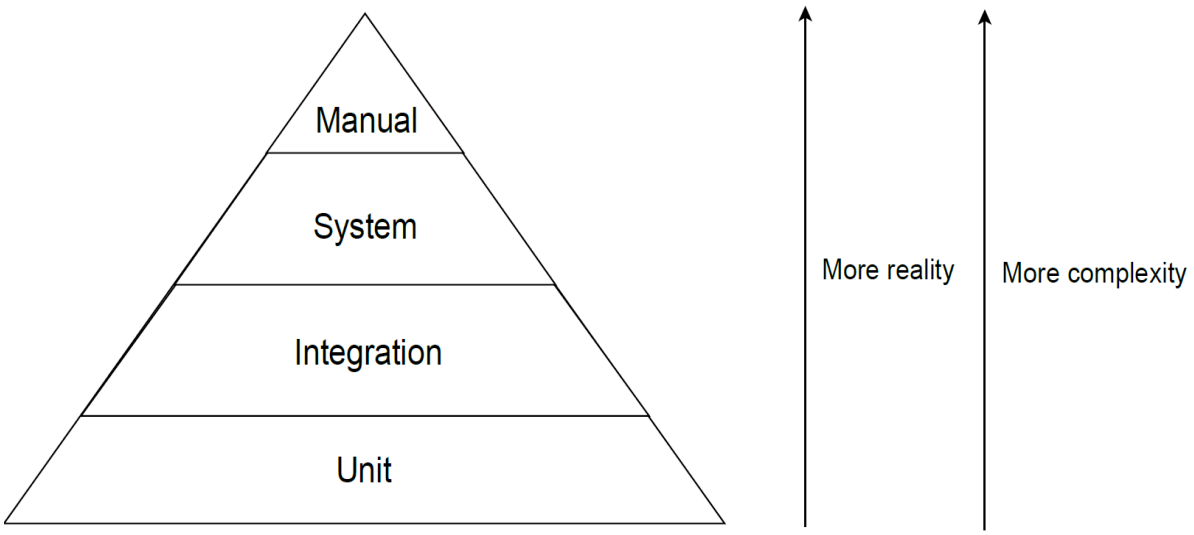


*Figure 2: The testing pyramid*

### 5.1 Testing pyramid

The diagram indicates all the test levels we discussed, plus manual testing. The more you go to the top, the more real the test is however, the more complex it is to devise it.

**How much should we do for each level then?** The common practice in industry is also represented in the diagram. The size of the pyramid slice represents the amount of test we would want for each test level. Unit tests are at the bottom of the pyramid, which means that we want most tests to be unit tests. The reasons for this have been discussed in the unit testing section (they are fast and require less effort to be written). Then, we go for integration tests, of which we do a bit less than unit tests. Given the extra effort that integration tests require, we make sure to write tests for the integrations we indeed need. Lastly, we perform a bit less system tests, and even fewer manual tests.

Again, note that the further up the pyramid, the closer to reality (and more complex) the tests become. These are important factors in determining what kind of tests to create for a given software system. We will now go over a couple of guidelines you can use to determine what level of tests to write when testing a system (which you should take with a grain of salt; after all, software systems are different from each other, and might require specific guidelines):

We start at the unit level. When you have implemented a clear algorithm that you want to test, unit tests are the way to go. In an algorithmic piece of code, the parameters are easily controllable, so with unit test you can test the different cases your algorithm might encounter.

Then, whenever the system is interacting with an external component, e.g. a database or a web service, integration testing is the way to go. For each of the external components that the system uses, integration tests should be created to test the interaction between the system and the external component.

System tests are very costly, so often we do not test the entire system with it. However, the absolutely critical parts of the system (the ones that should never, ever, stop working), system tests are fundamental.

As the testing pyramid shows, the manual tests are not done a lot compared to the other testing levels. The main reason is that manual testing is expensive. Manual testing is mainly used for exploratory testing. In practice, exploratory testing helps testers in finding issues that could not be found any other way.

Something you should definitely try to avoid is the so-called ice cream cone. The cone is the testing pyramid but put upside down.

### 5.2 Ice cream cone (Testing Cone)

It is common to see teams mostly relying on manual tests. Whenever they have system tests, it is not because they believe in the power of system tests, but because the system was so badly designed, that unit and integration tests are just impossible to be automated. In other words, hose who apply the testing pyramid try to avoid the so-called ice-cream cone anti-pattern.

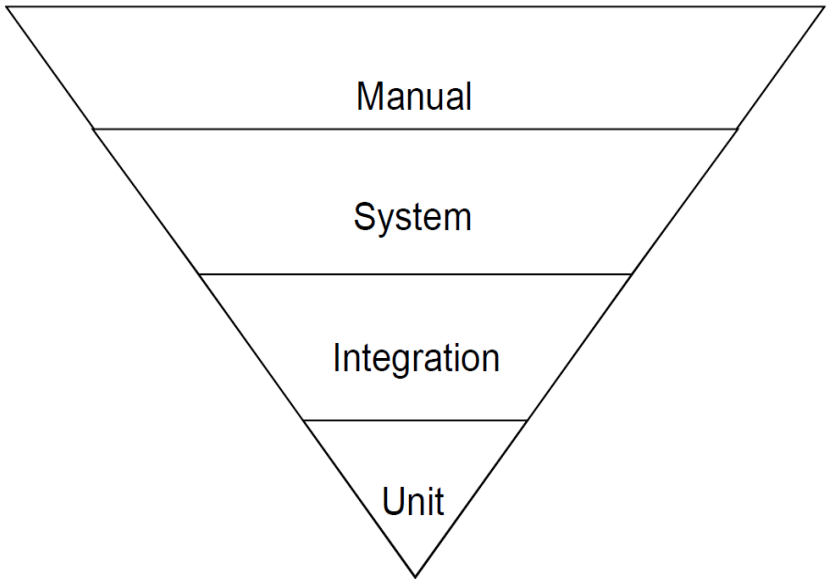


*Figure 3: The Testing Cone*

Imagine the testing pyramid upside down. In this new version, manual testing has the largest area, which means more effort on manual testing.

As mentioned earlier, our primary focus in this unit will be on testing functional correctness, at the unit, integration, and system levels. In practice, most programs are tested using both white-box and black-box techniques. Next, we are talking about Principles of Testing.

## 6. Principles of Testing

A simplistic view on software testing is that, if we want our systems to be well-tested, all we need to do is to keep adding more tests until it is enough. We wish it were that simple.

Indeed, a very important part of any software testing process is to know when to stop testing. After all, resources (e.g., money, developers, infrastructure) are limited. Therefore, the goal should always be to maximise the number of bugs found while minimising the amount of resources we had to spend on finding those bugs. Creating too few tests might leave us with a software system that does not behave as intended (i.e., full of bugs). On the other hand, creating tests after tests, without proper consideration, might lead to ineffective tests (besides costing too much time and money).

Given resource constraints, exhaustive testing is impossible - It might be impossible even if we have unlimited resources. Imagine a software system that has just 300 different flags (or configuration settings). Those flags can be set to either true or false (Booleans) and they can be set independently from the others. The software system behaves differently according to the configured combination of flags. This implies that we need to test all the possible combinations. A simple calculation shows us that 2 possible values for each of the 300 different flags gives $2^{300}$ combinations that need to be tested. As a matter of comparison, this number is higher than the estimated number of atoms in the universe. In other words, this software system has more possible combinations to be tested than the universe has atoms.

Given that exhaustive testing is impossible, software testers have to then prioritise the tests they will perform. When prioritising the test cases, we note that bugs are not uniformly distributed. Empirically, we observe that some components in some software systems present more bugs than other components.

Another crucial consequence of the fact that exhaustive testing is impossible is that, as Dijkstra used to say, program testing can be used to show the presence of bugs, but never to show their absence. In other words, while we might find more bugs by simply testing more, our test suites, however large they might be, will never ensure that the software system is 100% bug-free. They will only ensure that the cases we test behave as expected.

To test our software, we need a lot of variation in our tests. For example, we want variety in the inputs when testing a method, like we saw in the example above. To test the software well, however, we also need variation in the testing strategies that we apply.

Indeed, an interesting empirical finding is that if testers apply the same testing techniques over and over, they will at some point lose their efficacy. This is described by what is known as the **pesticide paradox** (which nicely refers to "bugs" as an equivalent term for software faults): *"Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual."* In practice, this means that no single testing strategy can guarantee that the software under test is bug-free. A concrete example might be a team that

solely relies on unit testing techniques. At some point, maybe all the bugs that can be captured at unit test level will be found by the team; however, the team might miss bugs that only occur at integration level. From the pesticide paradox, we thus conclude that testers have to use different testing strategies to minimise the number of bugs left in the software. When studying the various testing strategies that we present in next chapters, keep in mind that combining them all might be a wise decision.

The context also plays an important role in how one devises test cases. For example, devising test cases for a mobile app is very different from devising test cases for a web application, or for software used in a rocket. In other words: testing is **context dependent**.

## 7. Manual & Automated Tests

Testing (or manual testing) is different from writing tests. Developers write tests as a way to give them space to think and confidence for refactoring. They both provide developers with different outcomes. The former gives confidence and support throughout development. The latter makes sure the software will really work in all cases. While they are both important, it's about time for both communities to get aligned. Both should be done.

- Developers to get more familiar with more advanced software testing techniques (which researchers have been providing). They are definitely a good addition to their seatbelts and will help them to better test their software.
- Educators to teach both perspectives. My perception is that universities teach testing techniques in an abstract way (developers need to learn how to use the tools!), while practitioners only teach how to use the automated tools (developers need to learn how to properly test!).
- Researchers to better understand how practitioners have been evolving the software testing field from their perspective as well as to better share their results, as developers do not read papers. Research lingo is for researchers.

For the first six weeks we focus on manual testing using black box and white box testing techniques and apply them on the given artefact to write test cases and identify bugs. From week 7 onwards, we shall focus on a test-driven development approach and use it for assignment 2.

## 8. Some important concepts

We shall discuss some important concepts and terminologies which we are using in next weeks.

### 8.1 What is a Test Case?

In its most basic form, a test case is a simple pair of <input, expected outcome>. If a program under test is expected to compute the square root of nonnegative numbers, then four examples of test cases are as shown in Figure 4

$TB_1$: $< 0, 0 >$,
$TB_2$: $< 25, 5 >$,
$TB_3$: $< 40, 6.3245553 >$,
$TB_4$: $< 100.5, 10.024968 >$.

*Figure 4: Example of basic test cases*.

In a basic sense the outcome of program execution is a value(s) produced by the program – that could be the output of the local observation or manipulation. The outcome could also be a state change due to the execution of a program or an updation, deletion operations (applied in case of a database).

### 8.2 Testing Activities

In order to test a program, a test engineer must perform a sequence of testing activities. Most of these activities have been explained in the following. These explanations focus on a single test case – for multiple test cases, these steps can be repeated.

**Identify an objective to be tested:** The first activity is to identify an objective to be tested. The objective defines the intention, or purpose, of designing one or more test cases to ensure that the program supports the objective. A clear purpose must be associated with every test case.

**Select inputs:** The second activity is to select test inputs. Selection of test inputs can be based on the requirements specification, the source code, or our expectations. Test inputs are selected by keeping the test objective in mind.

**Compute the expected outcome:** The third activity is to compute the expected outcome of the program with the selected inputs. In most cases, this can be done from an overall, high-level understanding of the test objective and the specification of the program under test.

**Set up the execution environment of the program:** The fourth step is to prepare the right execution environment of the program. In this step all the assumptions external to the program must be satisfied.

**Execute the program:** In the fifth step, the test engineer executes the program with the selected inputs and observes the actual outcome of the program.

**Analyse the test result:** The final test activity is to analyse the result of test execution. Here, the main task is to compare the actual outcome of program execution with the expected outcome. The complexity of comparison depends on the complexity of the data to be observed. The observed data type can be as simple as an integer or a string of characters or as complex as an image, a video, or an audio clip. At the end of the analysis step, a test verdict is assigned to the program. There are three major kinds of test verdicts, namely, pass, fail, and inconclusive.

### 8.3 Test Report

A test report must be written after analysing the test result. The motivation for writing a test report is to get the fault fixed if the test revealed a fault. A test report contains the following items to be informative – it may vary based on the complexity of the test scenario:

- Explain how to reproduce the failure.
- Analyse the failure to be able to describe it.
- A pointer to the actual outcome and the test case, complete with the input, the expected outcome, and the execution environment.

### 8.4 Traceability Matrix

A traceability matrix is generated to make an association between requirements and test objectives to provide the highest degree of confidence. Basically, a traceability matrix tells us the test cases that are used to verify a requirement and all the requirements that are partially verified by a test case.

Before going into further details let us look at few more very important concepts. You must be familiar with the following error screens. Are you?

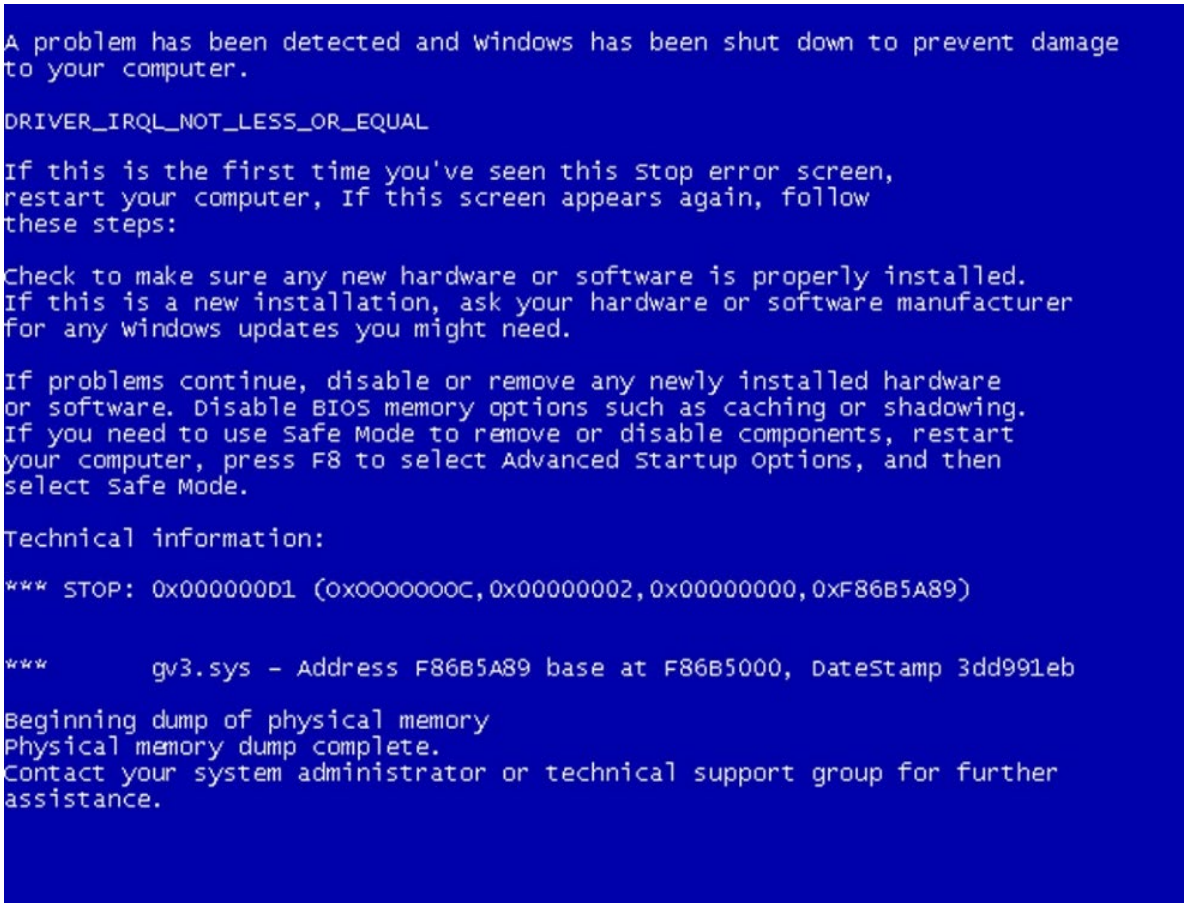


*Figure 5: A typical Windows Error Blue Screen*

And if you are a Windows 10 user, something like this…We have talked about it last week.

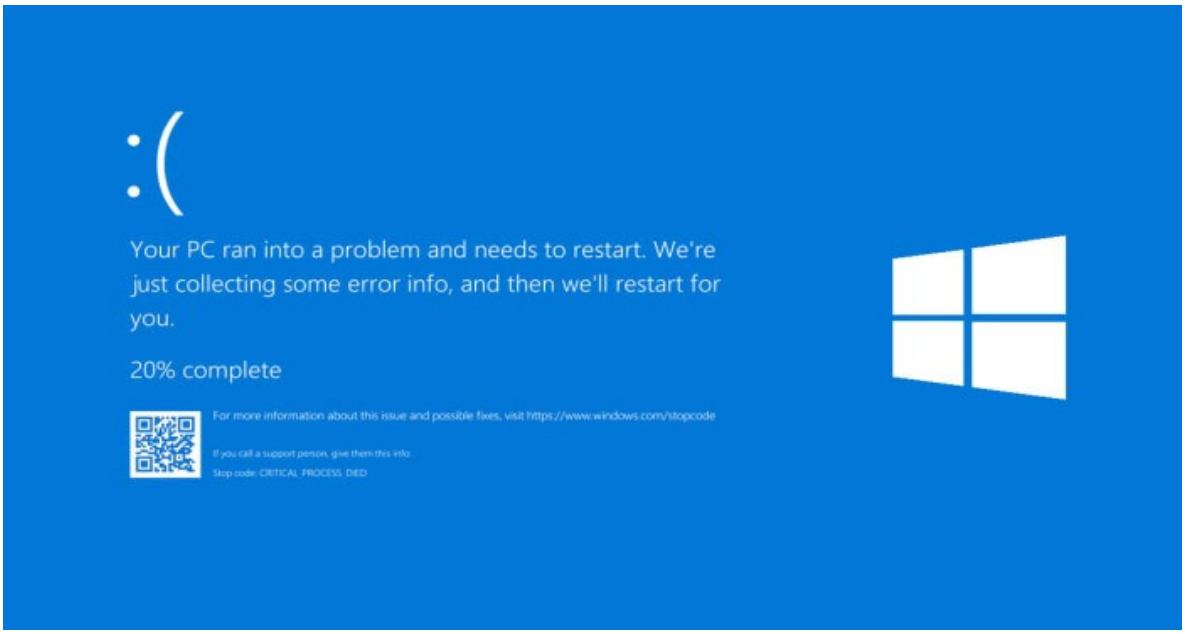


*Figure 6: An error screen for Windows 10.*

Why so? Well! Because there is an error in the system and the user must be notified of it and a possible solution, if any, could also be provided. Here the question arises what are failures, errors, and defects?

### 8.5 Error, Mistake, Failure, Defect, Bug, Debug, Fault...are they the same?

Key to the correctness aspect of software quality is the concept of defect, failure, fault, and Error. These words all look the same but there is a minor difference among all. Let us explore them:

**Failure:** A component or a system behaves in a way that is not expected...so this means a user expects a system to behave in an expected way, but it behaves differently. A common example is a blue screen shown above. Instead of having a result you expected such as you install a game and expect it to run, instead the system crashes - this is what the **failure** is. In a Software Engineering jargon, the term **failure** refers to a behavioural deviation from the user requirement (expectation - in other words) or the product specification.

**Defect:** A failure is usually caused by a defect so a **defect** can be a flaw in a component that can cause a system to behave incorrectly e.g., an incorrect statement. In other words, a defect, if encountered during execution, may cause a failure. **Bug** and **Fault** are synonyms for **Defects** and have the same meanings. It is possible that a bug exists in the code, but a failure never happens...interesting? Yes! perhaps the statement is never executed. So it will become a failure once it goes to a final user and the system behaves incorrectly for example a comparison in an if statement that uses a "less than" operator *(<)* instead of a "greater than" operator *(>)*. A broken connection is an example of a hardware fault.

**Error (Mistake):** A human action that produces an incorrect result. Perhaps, while developing a software, a developer fails to recognise a corner case and introduces a bug that results in a failure. Let us take an example. You are developing a travel agent software and while executing the code the system generates a *ClassCastException* (one of the types of exceptions in Java). So, what exactly happened? A travel information **fails** to deliver the required information, caused by a **fault** in a code introduced by **mistake** by a developer. In other words: a mistake by a developer can lead to a fault in the source code that will eventually result in a failure.

**Debug or Debugging:** The term **debug** general means "*get rid of the bugs*". Sometimes, it also includes activities related to detecting the presence of bugs and dealing with them. In this unit, we are not discussing debugging, or any activities related to it. We only mention it for the sake of the definition and its relatedness to the terminologies discussed above.

It is important to know these definitions as later these concepts will help us in understanding the basic testing techniques, that we learn in much detail in coming weeks. In the next week's notes, we will introduce several black-box test selections techniques and discuss how to apply them.

## Summary

- Testing is to conform the expected behaviour of the program

- Testing should be functionally correct and can have additional objectives such as reliability robustness etc.
- Unit, integration, and system are three major types of testing.
- Testing Pyramid
- Avoid Ice Cream Cone
- Testing is contextual based, and much testing should be performed to reduce the number of bugs.
- Both manual and automated testing should be done to improve the software quality.
- Failure: A failure is said to occur whenever the external behaviour of a system does not conform to that prescribed in the system specification.
- Error: An error is a state of the system. In the absence of any corrective action by the system, an error state could lead to a failure which would not be attributed to any event subsequent to the error.
- Fault: A fault is the adjudged cause of an error.

# References


1. Chapter 2 of the Foundations of software testing: ISTQB certification. Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans, Cengage Learning EMEA, 2008.
2. Vocke, Ham. The Practical Test Pyramid (2018), https://martinfowler.com/articles/practical-test-pyramid.html.
3. Fowler, Martin. TestingPyramid (2012). https://martinfowler.com/bliki/TestPyramid.html
4. Software Testing: From Theory to Practice accessible at https://sttp.site/
5. Testing vs Writing Tests accessible at https://medium.com/@mauricioaniche/testing-vs-writing-tests-d817bffea6bc
6. Software Testing and Quality Assurance Theory and Practice. Kshirasagar Naik and Priyadarshi Tripathy