**Nomes:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Exercício – Inspeção**

Vocês são responsáveis pela equipe de QA de uma empresa de desenvolvimento de software. Vocês precisam convencer a gerência da empresa de que teste de software deve ser priorizado e para isso, solicitaram que um dos testadores organizassem um documento para explicar como teste de software funciona. O papel de vocês agora é **inspecionar** este documento, na perspectiva de testador, procurando possíveis defeitos no documento. Para isso, utilize o seu conhecimento sobre teste de software e, como apoio, o TOB-SST.

Os defeitos encontrados devem ser inseridos na tabela a seguir. Quando encontrar um defeito no texto, sublinhar o texto e colocar um número para associá-lo a tabela fornecida (na coluna “Identificação do Defeito”). Em “Descrição”, explicar porque você considera isso como um defeito e qual seria o correto. \*O número de linhas da tabela não significa o total de defeitos\*.

**Tabela de Defeitos Encontrados**

Tipo do Defeito: Omissão (O), Informação estranha (IE) , ambiguidade(A), inconsistência(I) e fato incorreto (FI)

|  |  |  |
| --- | --- | --- |
| **Identificação do defeito** | **Tipo do**  **Defeito** | **Descrição** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**Document to be revised**

Software testing consists of the *dynamic* verification that a program provides *expected* behaviors on an *infinite* set of test cases, suitably *selected* from the usually finite execution domain. However, one of the most important limitations of software testing is that testing can show only the presence of failures, not their absence. This is a fundamental, theoretical limitation; generally speaking, the problem of finding all failures in a program is undecidable. Testers often call a successful (or effective) test one that finds an error.

In software testing, the following terms are employed:

* *Software Fault:* A static defect (or bug) in the software.
* *Software Error:* An incorrect internal state that is the manifestation of some fault.
* *Software Failure:* External, incorrect behavior with respect to the requirements or other description of the expected behavior.

To illustrate, consider a medical doctor making a diagnosis for a patient. The patient enters the doctor’s office with a list of *failures* (that is, *symptoms*). The doctor then must discover the *fault*, or root cause of the symptom. To aid in the diagnosis, a doctor may order tests that look for anomalous internal conditions, such as high blood pressure, an irregular heartbeat, high levels of blood glucose, or high cholesterol. In our terminology, these anomalous internal conditions correspond to *errors*. While this analogy may help us thinking about faults, errors, and failures, software testing and a doctor’s diagnosis differ in one crucial way. Specifically, faults in software are *design mistakes*. They do not appear spontaneously, but rather exist as a result of some (unfortunate) decision by a human.

For a more technical example of the definitions of fault, error, and failure, consider the following method:

public static int numZero (int[] x) {

*// Effects: if x == null throw NullPointerException*

*// else return the number of occurrences of 0 in x*

int count = 0;

for (int i = 1; i < x.length; i++)

{

if (x[i] >= 0)

{

count++;

}

}

return count;

}

The error in this program is that it starts looking for zeroes at index 1 instead of index 0, as is necessary for arrays in Java. For example, numZero ([2, 7, 0]) correctly evaluates to 1, while numZero ([0, 7, 2]) incorrectly evaluates to 0. In both of these cases the fault is executed. Although both of these cases result in an error, only the second case results in fault. To understand the error states, we need to identify the state for the program. The state for numZero consists of values for the variables x, count, i, and the program counter (denoted PC). For the first example given above, the state at the if statement on the very first iteration of the loop is ( x =[2, 7, 0], count = 0, i = 1, PC = if). Notice that this state is in error precisely because the value of i should be zero on the first iteration. However, since the value of count is coincidentally correct, the error state does not propagate to the output, and hence the software does not fail. In other words, a state is in error simply if it is not the expected state, even if all of the values in the state, considered in isolation, are acceptable. More generally, if the required sequence of states is *s*0*, s*1*, s*2*, . . .* , and the actual sequence of states is *s*0*, s*2*, s*3*, . . .* , then state *s*2 is in error in the second sequence. In the second case the corresponding (error) state is (x=[0, 7, 2], count=0, i=1, PC = if). In this case, the bug propagates to the variable count and is present in the return value of the method. Hence a failure result.

A practical problem associated with software testing is how to provide the right values to the software and observing details of the software’s behavior. A test case is the combination of input value and observed result necessary for a complete execution and evaluation of the software under test.A test set is a set of test cases and the testing coverage is a property of a test set, rather than a property of a single test case. A test case is composed by the values, expected conditions and the outputs produced by the program.

No matter what coverage criterion is used, sooner or later one wants to know whether a given program executes correctly on a given input. This is the oracle problem in software testing.

The oracle problem can be surprisingly difficult to solve, and so it helps to have a range of approaches available.

One of the aims of testing is to reveal as many failures as possible. Many tools have been developed to do this (functional, structural, fault-based). These techniques attempt to “break” a program by being as systematic as possible in identifying inputs that will produce representative program behaviors; for instance, by considering subclasses of the input domain, scenarios, states, and data flows. The classification of testing techniques is based on how tests are generated: from the software engineer’s intuition and experience, the specifications, the code structure, the real or imagined faults to be discovered, models, or the nature of the application. One category deal with the combined use of two or more techniques.

Sometimes these techniques are classified as structural testing and functional testing:

* *Structural testing:* Deriving tests from external descriptions of the software, including specifications, requirements, and design.
* *Functional testing:* Deriving tests from the source code internals of the software, specifically including branches, individual conditions, and statements.

Software testing is normally conducted in phases, as software is developed. The three main phases are:

* *Unit Testing*: this phase verifies the functioning in isolation of software elements that are separately testable. Depending on the context, these could be the individual subprograms or a more significant component made of highly cohesive units;
* *Integration Testing*: this phase occurs after the units are tested individually, and aims to ensure that the interactions among software components are working correctly;

A test selection criterion is a means of selecting test cases or determining that a set of test cases is sufficient for a specified purpose. Test adequacy criteria can be used to decide when sufficient testing will be, or has been accomplished.

A well-known functional testing criterion is equivalence partitioning. This criterion involves partitioning the input domain into a collection of subsets (or equivalent classes) based on a specified criterion. This criterion may be different computational results, a relation based on control flow or data flow, or a distinction made between valid inputs that are accepted and processed by the system and invalid inputs, such as out of range values, that are not accepted and should generate an error message or initiate error processing. A representative set of tests (sometimes only one) is usually taken from each equivalence class. The idea is that each test case represents all input domain that equivalency class.

An important aspect is the software techniques are complementary; in fact, they use different sources of information and have been shown to highlight different kinds of problems. They could be used in combination, depending on budgetary considerations.