

Program testing

Goal: elimination of errors (it's estimated that 50% of resources are used in this task).

It is advisable to start by distinguishing the techniques of formal verification (*proving*) from the ones of testing. The former are used to formally prove that the program satisfies the specification. The latter ones work by executing the program with different data sets, as complete as possible, in order to evaluate the program correction.

Criticism of these methods: formal verification techniques are complex and, therefore, they must be performed by quite qualified personnel, hence their application being expensive and complex.

Besides being expensive, formal verification just proves that the program satisfies the specification, and therefore, it will just guarantee the absence of errors when the specification does it. In other words, it is necessary that we can guarantee that the specification is complete and correct enough. In spite of this, in some countries of Europe the law obliges to perform formal verification of the more critical programs (such as the ones managing the reactors of nuclear power plants).

Program testing is the most used error debugging method in the industry in spite of seeming, and maybe being, theoretically the less reliable. The reasons are the lower cost and the fact that they can more easily adapt to the needs of each specific case. It is important to remark that they can be easily made simpler or more complex depending on the importance given to the program reliability.

The problem about these testing techniques is that in fact they can only guarantee that the program works correctly with the data and in the conditions that have been tested, and not the complete absence of errors, even though one tries that the data

Other goals. Besides error elimination, with program testing one wants to check also that the program functionality is complete (that nothing is missing) and that NFR (non-functional requirements, which can include conditions about performance like efficiency, etc.) are satisfied.

Error debugging. In this phase of the program development not only the possible errors must be found, but also their cause must be located and they must be eliminated, that is, they must be debugged.

Debugging also means that when one error is detected it must be found out in which phase of the life cycle it was originated, correct what's necessary and see what other implications it will have in subsequent phases, which will have to be corrected as well if necessary.

We call *test set* to the set composed by both the data used to perform the program testing (including the input peripherals manoeuvres, such as the mouse ones) and the expected results (including the effects that may take place).

For each program execution with a test set, the obtained results must be evaluated and compared with the expected ones. It will be necessary to treat the eventual

differences between a result (the one obtained) and the other (the expected one) as an error.

It's essential to take into account that each time the program is modified the testing process must begin from scratch, as if a new program had been written.

Watch out! Real time program testing presents a specific set of problems, since in this case things get more complex because of the difficulty to simulate reality, specially as to the coincidence of events.

Classifications can be done according to different criteria. First one we can consider is according to who is the performer of the test:

- The programmer
- The designer
- The analyst
- The customer or user

We can see that the two former ones know the algorithms quite well, while the two latter will just know the expected behaviour, the functionality.

According to the used technique:

- *White box*: feasible only when algorithms are known. It is about causing the execution, at least once, of every single statement, this means passing through all the branches of the alternative statements (*if* and others) and executing or not the loops.
- *Black box*: these tests are independent of the structure of the algorithm, they are based only on the expected behaviour. It is about stating the sets of data that will produce a uniform behaviour, this defines an equivalence relationship (a test set is equivalent to another if it makes the program behave the same way) that allows to establish a partition of the set in equivalence classes. Then one element from each class must be chosen in order to perform the tests (and this set of elements is called *quotient set*).

Considering that the complete set of test sets would be composed by all the possible combinations of data and peripheral manoeuvres, we say that a set of test sets is *minimally complete* in white box if it accomplishes the execution of all the statements. In black box, if it is a quotient set (which contains a representative of each uniform behaviour class). These sets of minimally complete test sets do not have to be the same, that is, the white box ones might be different from the black box ones.

According to the testing unit, the test can be performed:

- On a *component* (in general components are routines, operations, functions, etc. In object orientation (OO) the component will always be a class, never will an element or smaller module be tested).
- On part of the components of the program during the *integration* process.
- On the program or complete system for the *validation* of the correctness of the product by the developer.
- On the complete program or system for the *acceptance* of the finished product by the customer (or user).

Validation will be done, in principle, at the operative development environment with the criteria of the developer (which are the specification), whereas the acceptance

will be always done at the customer's house, at his/her operative environment and with his/her criterion (based on what he/she wanted or needed, it will normally be *beta tests*). *Alfa tests* are those ones which are performed with the attendance or closeness of the programmers or technicians who have developed the program, whereas *beta tests* are those ones where technicians are not within reach.

In any test, it must be defined:

- Unit of the test (object of the test: a component, the integration of several ones, etc.)
- Technique to be used (white or black box).
- Performer of the test (user, analyst...)
- Test set, which must be at least minimally complete.

Integration testing: it consists of testing together all the components already tested individually. This can be performed with bottom-up, top-down or mixed criteria.

Bottom-up means starting from the lowest components of the program structure, elements that do not aggregate any others and that are designed to aggregate into others. One goes further by going up the levels in the structure and composing structures which are more and more complex until you get to have the whole program.

Top-down criterion works the other way round. One starts from the component that comprises the whole program and proceeds by incorporating other components of the lower level (going down) that are part of its structure until getting to integrate the simplest ones.

Mixed criteria start from a component, which may be anyone, and alternatively use both previous criteria as needed. We always end up with the complete program or system.

When working bottom-up, we must do, for each component (or integrated group of components), a program that allows to access its operations (or functionalities), that is, a program that allows to make the appropriate calls to execute them. These elements are called *drivers*, and they are simple programs, sometimes quite rudimentary, specialized in being able to use the functionalities of the class or set of classes being tested.

When working top-down, we will have to replace the components composing the tested class with elements that somehow emulate their behaviour of that simply allow us to know that the call to the still-not-integrated component has been done correctly. These elements are called *stubs*, they are simple programs sometimes quite rudimentary that allow us to know that we have correctly called that component. *Stubs* sometimes allow to introduce the answers by hand, sometimes they give fixed answers (constant values), sometimes they evaluate the precondition and postcondition of the called functionality, etc.

In OO the integration is done by following inheritances, aggregations, associations and dependencies, normally through the layers and inside the layers distinguishing controllers (domain or interface ones) from basic classes. The smallest element o test (or component) is a class, never smaller.

Debugging techniques: Normally they are based on the tracing of the execution of the program, and some languages, as Eiffel, allow to use the specification as well.

The process is usually started from the point where the erroneous (or unexpected) result was produced and goes backwards in the direction of the execution flow of the program. One can also try to determine the suspicious points.

During the debugging process we must be careful to avoid the production of new errors and always remember to review all the phases in order to detect the real cause.

Debugging tools: normally they are provided by the compiler or the development environment we work with, we stand out:

- *Debugging* programs, sometimes they are complex or difficult to use.
- CASE tools usually contain auxiliary debugging tools that can even comprise different phases of the life cycle.
- Use of the formal specification.

Form: remember that at the second submission you must document each test executable (many times it will be a *driver*) according to the standard test set description form of the subject. In this form those fields which do not have to be filled in can be left blank, but they are not eliminated. Remember also that you must include the compilation options used in order to generate each executable as well.

There will be three files with the same name at the folder where the executable is: the source code (.java), the executable (.class) and the form (.txt, .doc, .html, or htm) that documents the test.

A testing program must admit new data without having to compile again. That is, the data of the test sets must be outside the source code of the program. Obviously these data can be in easily editable files (for example in plain text format), it is not essential that the testing program is interactive.