Software Engineering: Testing workflow

Original version: Fulvio Sbroiavacca (https://tinyurl.com/sw-testing)

Tanslated and adapted by Mario Cimino

What is the purpose of the testing workflow?

- The purpose is not to show that the software is bug-free (it is unrealistic)
- The purpose is to find as many errors as possible
- For this purpose, the code should be tested by a different team (testing team) w.r.t. the team that produced it (implementation team)

Levels and operations of testing

Testing is carried out at various levels for distinct operations:

- **Unit test**
a unit is the smallest block of software that it makes sense to test, (e.g. a single low-level function that is checked as a stand-alone entity)

- **Module test**
a module is a collection of interdependent units

- **Subsystem test**
a subsystem is a significant aggregate of modules often designed by different teams; there may be interface problems that need to be fixed

- **System or integration testing**
it is the complete product test

Operations of testing

• **α-test**
- the system is developed for a small group of customers
- the system is placed in its final environment and tested with the data on which it will normally have to operate

• **β-test**
- the system is distributed to a community of users
- the system is tested by several users, who use it and provide their observations and the errors found

• **Benchmark**
- the system is tested on standardized data in the public domain for comparison with other equivalent products on the market
- it can be requested by contract

• **Stress testing**

- is checks how the system behaves when it is overloaded, bringing it to the limit
- the stress test allows you to cause an error and verify that the system fails in an acceptable way (fail-soft)

Testing and Debugging

- Testing and debugging are two different concepts
- **Testing** detects the **presence of errors**
- **Debugging** consists in the **localization of the error**, its **analysis** and its **correction**
- **Regression testing** is performed after debugging
- It verifies the behavior of the module that has been corrected w.r.t. the cooperating modules
- It makes sure the correction of the error has not introduced new errors

Error distribution

- Errors tend to concentrate in certain modules, generally the more "complex" modules

- A module with many errors should be checked carefully: it will probably contain more errors

- According to some statistical studies, the probability that there are other "latent" errors increases with the number of errors already found according to a "logistic" curve

Error distribution

- Probability of existence of other errors than those found

Errors introduced by maintenance

• In the **maintenance** activity, aimed at evolutive changes (new customer needs) and at correcting errors, **new errors** are introduced

• Many decisions initially made by programmers are not documented, they are not evident from the code; other decisions made later in the modification phase tend to be forgotten by authors and unknown to others

• To modify a part of code written by others, or simply "dated", with the aim of eliminating errors, easily leads to introduce further errors

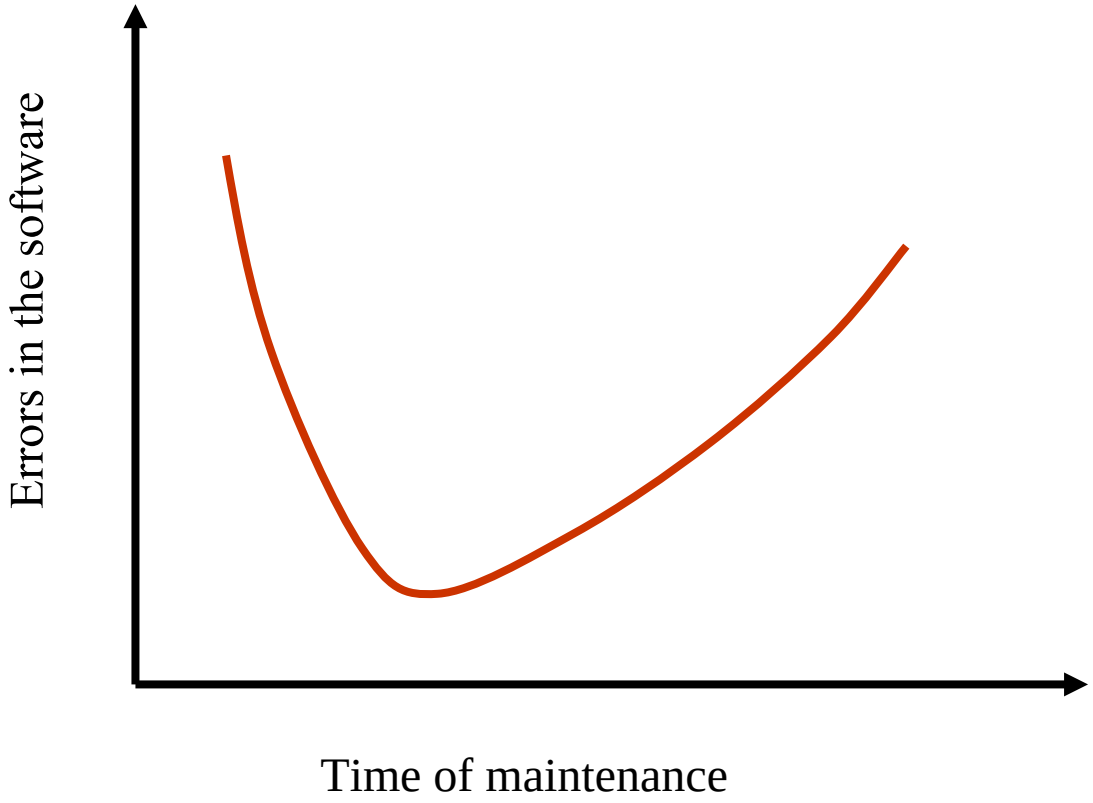
• The typical curve of errors as a function of maintenance time often tends to follow a close-to-**parabolic** curve

- initially there is a reduction in the number of errors; as maintenance time passes, continuous interventions tend to introduce lots of new errors

- When a module is subject to continuous maintenance, a redesign should be scheduled to optimize maintainability and extensibility

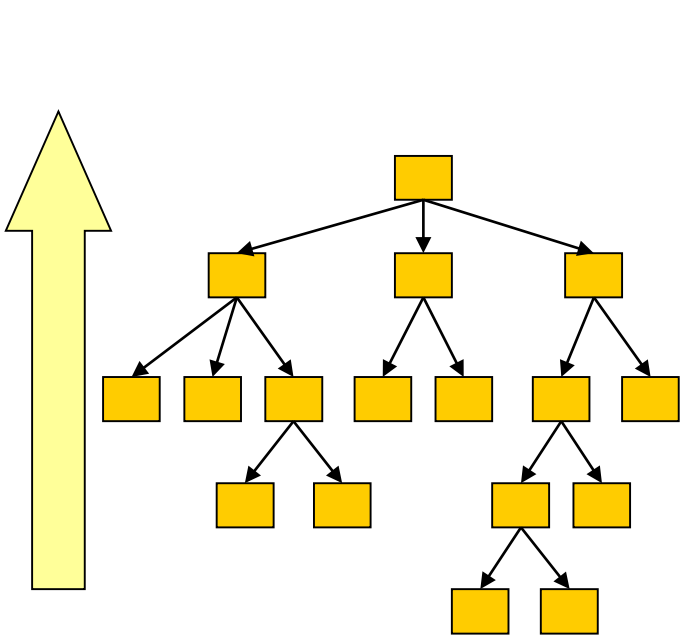
Error distribution

Errors introduced by maintenance

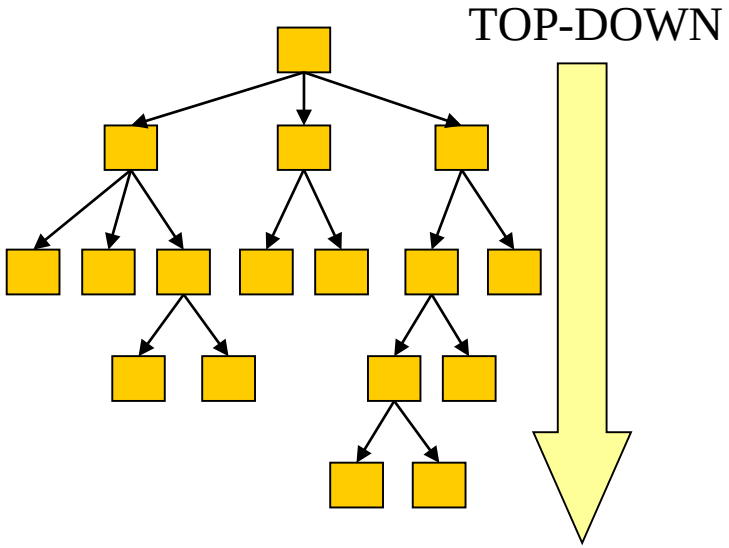


Testing strategy

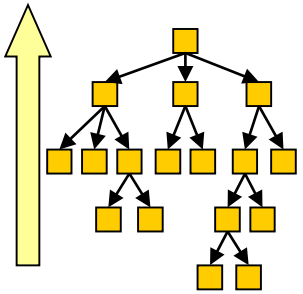
- Like design, testing can also be carried out top-down or bottom-up



BOTTOM-UP

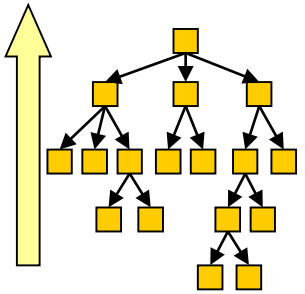


Bottom-up testing



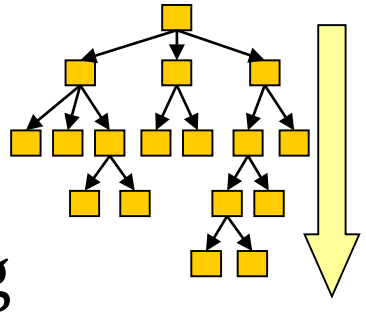
- First, the “lowest level” modules in the hierarchy produced by the design are tested; these are the smallest and most basic units of the program
- When the smallest components are correct, the next level of composition is tested, using the functionality of the previous level
- Progressively, the entire system is tested
- This approach has some issues:
  - **drivers** are needed, i.e. auxiliary software that simulates the call to a module by generating data to be passed as parameters
- - drivers replace the part of the code not yet integrated in the tested portion of the program: the top of the module hierarchy
-

Bottom-up testing: advantages and disadvantages



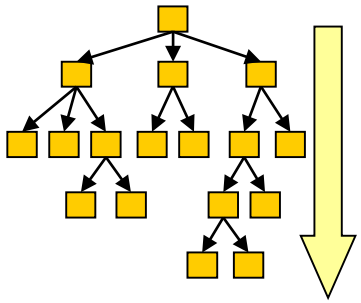
- The integration and testing process is intuitively **easier** to implement
- The later a bug is discovered, the more expensive it is to fix it, as it requires fixing it usually with a leaf module, and its partial re-testing, up to the modules where the bug was found

Top-down testing



- Initially, the module corresponding to the root is tested, without using the other modules of the system
- **Simulators**, called **stubs**, are used instead of the other modules of the system
  – these are elements that can return random results
  – request the data from the tester
  – can be a simplified version of the module
- After testing the root module of the hierarchy, its direct children are integrated
- – simulating their descendants by means of stubs
- This continues until all the leaves of the hierarchy are integrated.

Top-down testing: advantages and disadvantages



- The costliest errors to fix are discovered first
- This is particularly useful if the design is also top-down and the design, implementation and test workflows partially overlap:
  – a design error at the top of the hierarchy can be discovered and corrected before the bottom is designed
- An incomplete but functioning system is available at all times
- Stubs can be **expensive** to create

What is the best testing strategy?

- A compromise solution is usually adopted between the two strategies, to mitigate the drawbacks that each solution presents
- It is not convenient to integrate as many modules as possible at each step

  – for example, in a bottom-up strategy, testing the children of a module and then integrating them all together with the parent to test: this leads to broken code, without a clear idea of where the error is
- It is convenient to **introduce the tested modules one at a time**

  – this way it is possible to locate errors better
- In general it is more convenient to carry out **frequent and short tests**

Static checks

- The critical point of the test is represented by the **cost**
  – to reduce it, to improve the relationship between the cost and the power of the test, or to support the test with other verification techniques
- One form of static verification is **code inspection**
- It allows to identify between 60% and 90% of errors that would otherwise only be found during testing with often higher costs
- Inspection can also be performed by a specialized team, having the specification document, whose task is to report the errors to the programmers, who will then fix them
- The coding environment normally provides automatic tools that analyze the code indicating errors (for example, a call to a function with the wrong number of parameters)

Testing

- Code inspection alone is obviously not enough
  – it does not intercept errors due to the execution of programs
- The modules have to be run on **data samples**, observing the resulting behavior
- Testing has two problems
  – It is **never exhaustive**, therefore it does not prove the correctness of the code
  – It is **expensive**, in terms of machine use and human time

Test-cases

- To carry out a test it is necessary to develop a set of **test-cases**, each including:
  – **input** data (test-data) for the module to be tested
  – **description of the function internal** to the module
  –**output** expected from the function
- A test-case represents a characteristic, a requirement, conforming to the specifications and it is used as a unit to construct a test
- It is completed by a set of metadata
  – status (approved, rejected, modified)
  – context
  – link to specification
- – …

How do you build a test case?

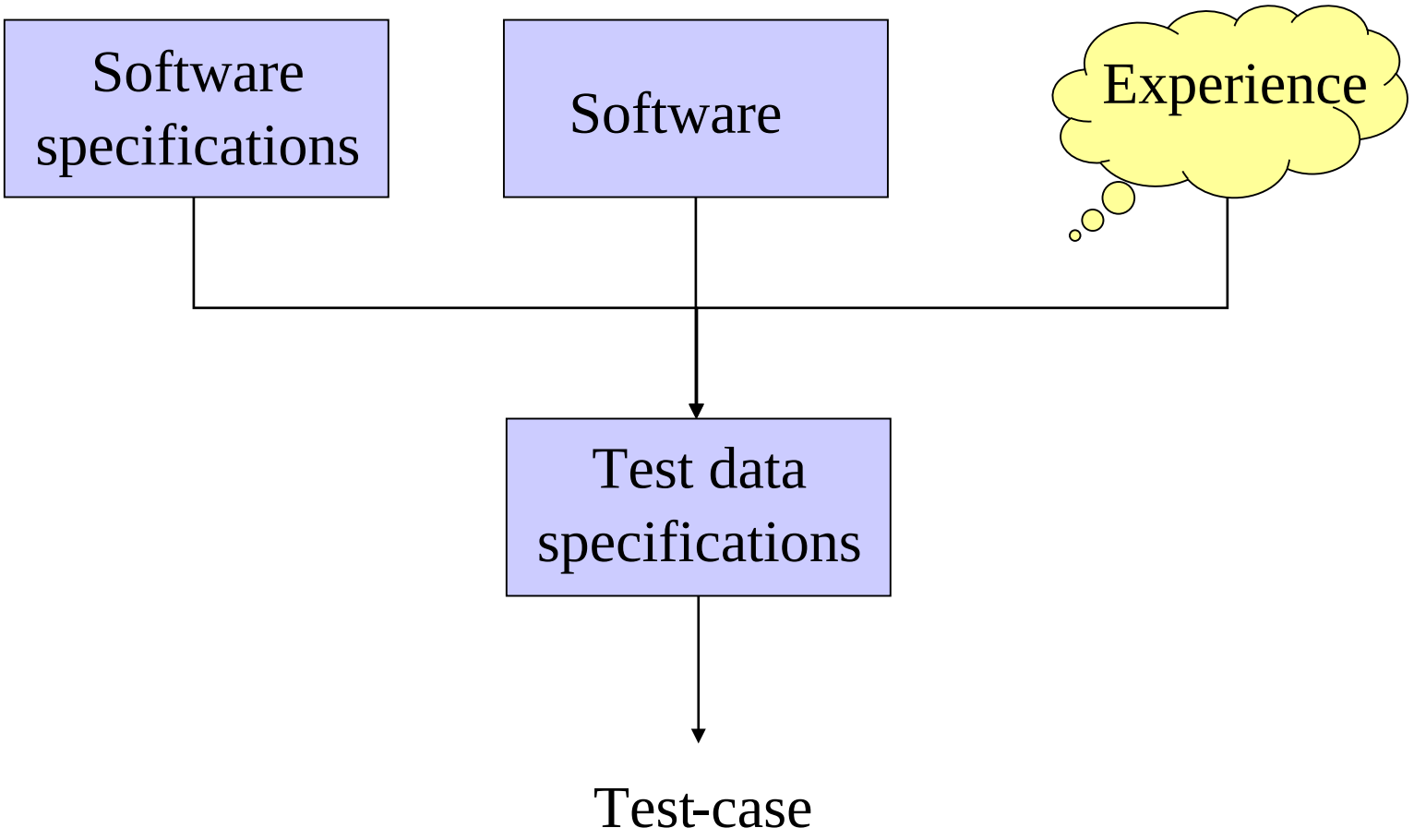
- To build a test-case it is necessary to know the **correct output** for a certain input of the function under examination
- **It should be taken from the specifications and documents produced by the design workflow**
- At the end of the input execution, the output is stored with the test-case and is compared with the expected output
- The test must be **repeatable**
  – for the test phases that occur later in the life of the software

How to choose the input?

- The first idea is to generate the input **randomly**
- This technique has the disadvantage of not being very uniform for small samples

  – to have reliability you need to generate a lot of test-data
- Better input selection criteria are needed
- The test team has to decide the characteristics of the test data on the basis of:
- – the **program**,
- – the **specifications**,
- – the **experience**

Test-case generation -



Testing methodologies

- **Black-box testing**

  – The specifications of the input data are achieved only from the specifications of the program

  – Without considering the code
- It is normally used for the first tests
- It has the advantage of being accessible to the customer (who does not see the code)

- **Structural testing**
- – You derive the specifications of the test data by looking at the code
- It is generally better than the black-box testing

Black box testing

- The choice of input data is based only on the specifications of the program

  – each function is matched with a more or less formal description of its correct inputs, the output that should be emitted and the function itself

  – for example, a function that returns the date accepts as input the day, the month in the respective ranges of 1:31, 1:12
- The idea of black-box testing is to **partition the set of admissible inputs into equivalence classes** so that within each class, each data has the same testing power
- Experience determines the selection criteria:

  – at least one value inside each class

  – all boundary values

Black-box testing and equivalence classes

- Some criteria for determining the equivalence classes
- For a variable v defined in an interval a…b there are three classes:

  $v < a$; $a \leq v \leq b$; $v \geq b$;
- For integer variables there is only one class
- – experience suggests to use also the value 0 and a negative value
- For alphanumeric strings, the determination of classes and boundary values depend on the restrictions on the format of the string

  – for example: the empty string, the one containing only characters, the one containing numbers, etc.
- One problem with this method
- – the combination of test data for all input components leads to a **huge number of test cases**

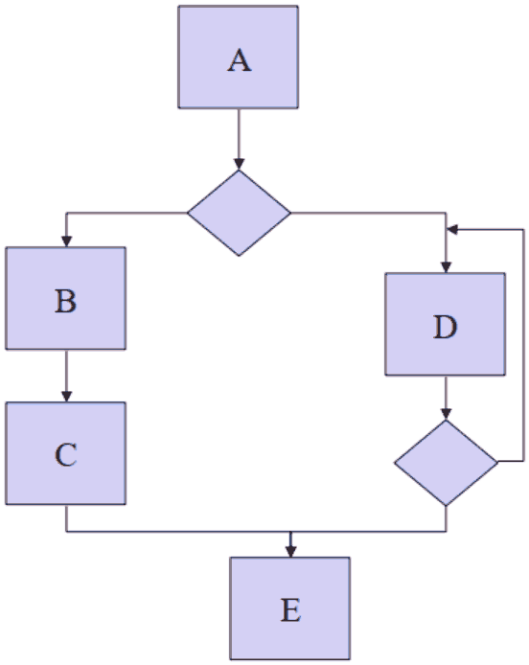
Structural testing

- The choice of input is based on the structure of the program
  – the control flow of the program is determined and expressed in the form of a flow-chart
- The idea of Structural Testing is to use test data that leads **to all possible computations**
- Data combinations are searched for all path combinations in the flow-chart
- In the case of cycles, at least two test data are taken
  – one who runs the loop
  – and one that doesn't do it

Types of Tests

example of structural test

- It is conducted according to the structure of the system
  – the flow of control is determined in the form of a flow-chart
  – using test data leading to all possible combinations of paths in the flow-chart
- E.g. it can be used in the test of web sites
  – applied to pages
  – verifying all the paths through the links



Automatic tools (Test Automation)

- Testing can be streamlined using **automated testing tools**

- These tools provide several supports, essential for the repeatability of the tests
  – Management of test cases
  – Test-data management
  – Storage of the tests carried out
  – Analytical reporting

- Automated testing cannot always fully replace manual testing
  – some checks are more effective if performed manually
  (verification of the graphic layout and usability of the product)

Test Automation cases

• **Functional tests** of a product
– need to repeat the same test many times
– different values of the input data
– different configurations

• **New product releases**
– check that the changes made have not introduced regressions

• Manual testing makes the process particularly expensive
– cost and time constraints lead to non-execution of tests compromising the quality of the product

• Automating testing drastically reduces its cost
– increases the initial design cost, which is however returned by the time savings in the numerous automatic re-runs of the test cases

– repetitive verification operations easily lead to human errors; the automatic test performs the same operations several times with accuracy and precision

Simulators

• The most difficult software to test concerns exceptional or dangerous situations: in these cases, simulators of the environment in which the software will operate are used

• A simulator is an auxiliary program
(which will then not be used at the end of the development process) that mimics the actions of another program or hardware or environment behavior:
– it is used to test products whose malfunction can cause damage or (e.g. a program to operate a nuclear reactor)
– also serves to test the system under particular load conditions (stress-testing), difficult to obtain if not in particular conditions of the final environment
• The construction of simulators obviously has a significant impact on the cost of the finished product