Software testing
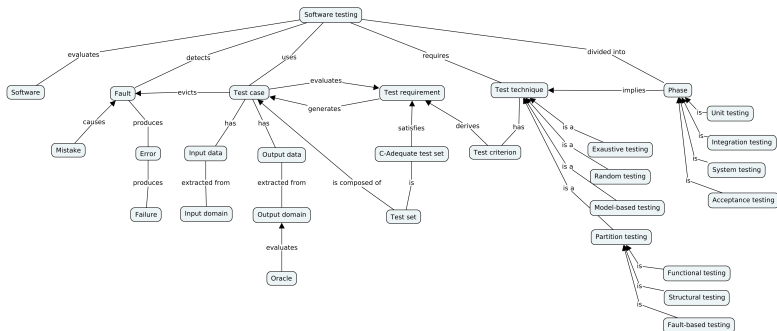## Fault-based testing

Marco Aurélio Graciotto Silva[1],   Ellen Francine Barbosa[2],
Márcio Eduardo Delamaro[2],   Auri Marcelo Rizzo Vincenzi[3],
José Carlos Maldonado[2]

[1]Federal University of Technology – Paraná (UTFPR) Campo Mourão, PR, Brazil

[2]University of São Paulo (USP) São Carlos, SP, Brazil

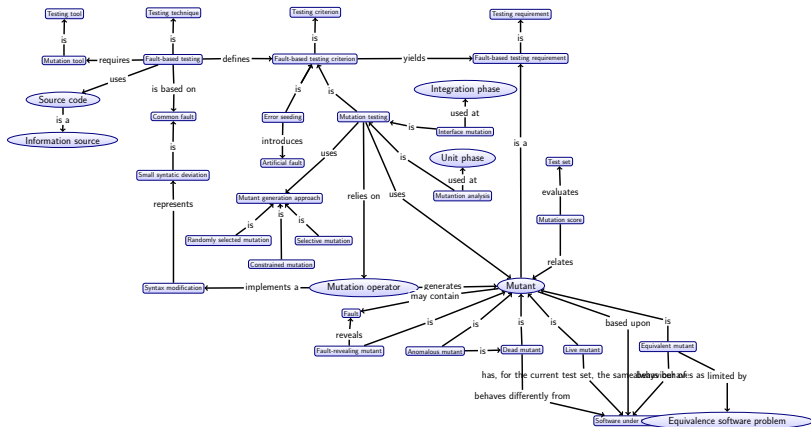[3]Federal University of São Carlos (UFSCar) São Carlos, SP, Brazil

Software testing

Fault-based testing:

Fault-based testing is a technique in which testing is based on historical information about common faults detected during the software development life cycle.

Rationale:

- Fault-based testing technique uses information about:
    - faults which are frequently found in software developments, and
    - specific types of faults that one may want to uncover.

Error seeding:

Error seeding introduces a known number of artificial faults in the program under testing before it is tested.

Rationale:

- Error seeding is based on the assumption that faults are uniformly distributed in the program.

How it works:

- After testing using the error seeding criterion, it is possible to estimate the number of remaining natural faults from the total number of faults found and the rate between natural and artificial faults.

Limitations:

- Error seeding requires programs that can support 10,000 faults or more in order to obtain a statistically reliable result.
- The artificial faults generated by error seeding may hide natural faults.
  - This is in general is not the case, as real programs present long pieces of simple code with few faults and small pieces of high complexity and with many faults.

Mutation testing:

Mutation testing is a fault-based test criterion that uses a set of products that differ slightly from product P under testing, named mutants, in order to evaluate the adequacy of a test set T.

Effectiveness:

- Mutation testing criterion is one of the most effective test criteria in detecting faults.
- It is frequently used to evaluate the quality of other testing criteria.
- It is frequently used to evaluate the efficacy of a test set.

Rationale:

- Mutation testing is based upon two principles:
  - The competent programmer hypothesis.
  - Coupling effect

Competent programmer hypothesis:

A good programmer writes correct or close-to-correct programs.

Coupling effect:

Complex errors are composition of simple ones.

Goal:

- The goal of mutation testing is to find a test set which is able to reveal the differences between a product P and its mutants (making them behave differently).

- The ideal situation in mutation testing would reveal all the mutants ead, which would indicate that the test set T is adequate for testing P.

Mutation testing basic procedure:

1. The tester should provide a product under testing P and a test set.

2. The program is executed against T and, if a failure occurs, a fault has been revealed and the test is over.

3. If no problem is observed, P may still have hidden faults that T is unable to reveal.
   - In this case, P is submitted to a set of mutation operators which transform P into a set of products called mutants of P.

4. Mutants are executed against the same test set T, generating either live mutants or dead mutants.

5. After executing the mutants and identifying the equivalent ones, an objective measure of test set adequacy is provided by the mutation score.

Limitations:

- The main problem with the mutation testing is the high number of generated mutants and, therefore, the high cost of executing a large number of mutants.
- Mutation testing faces the problem of deciding mutant equivalence which in general is undecidable.
- Mutation testing requires tester intervention in order to determine equivalent mutants.
  - Mutation testing requires good knowledge about the product implementation to ease the task of analyzing live mutants.

Mutant:

Mutation testing:

- In mutation testing, a single mutation is applied to the program P under testing.
  - Each mutant has a single syntactic transformation related to the original program.
- Tester has to create test cases that show that mutations create incorrect products.

Live mutant:

A live mutant is a mutant that has the same behavior of the product P for each test case T.

How to identify one?:

- A live mutant must be analyzed by the tester to check whether it is equivalent to product P or whether it can be killed by a new test case, thus promoting the improvement of the test set T.

Dead mutant:

A dead mutant is a mutant that has a diverse behavior from product P on at least one test case T.

Equivalent mutant:

A mutant M is said equivalent to a product P if, for all input data $d \in D$, $M(d) = P(d)$.

:

- An equivalent mutant is created by a mutation operation that does not result in a behavioral change for a product P, and, for every test datum in the input domain, P and the mutant always compute the same results.

Fault-revealing mutant:

A mutant M is a fault-revealing mutant to a product P if for any test case t such that $P(t) \neq M(t)$ we can conclude that $P(t)$ is not in accordance with the expected result

:

- The presence of a fault is revealed by killing the mutant.

Mutation score:

Mutation score is the relation between the number of mutants killed by the test set T and the difference between the total number of mutants and the number of mutants equivalent to P.

Meaning:

- Mutation score is an objective measure to evaluate the test set adequacy against mutation testing.
  - Mutation score ranges from 0 to 1.
  - The higher the mutation score, the more adequate is the test set.

Mutation operator:

Mutation operators are rules that define the changes to be carried out in a product P in order to create a mutant.

:

- Mutation operators model the most frequent faults or syntactic deviations related to a given programming language.

Objectives:

- Mutation operators are designed for a target language and should fulfill one of the following objectives:
  - Create a simple syntactic change based on typical errors made by programmers.
  - Force test cases to have a desired property (covering a given branch in the program, for instance).

Test strategy:

- Mutation operators can be selected according to the classes or faults to be addressed, allowing the creation of mutants to be done stepwise or even to be divided between testers working independently.

Mutation approaches:

- Different mutation approaches can be used to execute mutants against a test set and that may reduce the number of generated mutants.

Mutation approaches:

- Randomly selected mutation;
- Constrained mutation;
- Selective mutation.

24