Functional, Control and Data Flow, and Mutation Testing: Theory and Practice

Part I - More theory than practice

Auri M. R. Vincenzi[1], Márcio E. Delamaro[2], Erika N. Höhn[3] and José C. Maldonado[3]

[1]Mestrado em Informática
Universidade Católica de Santos

[2]Faculdade de Informática
Centro Universitário Eurípides de Marília

[3]Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo

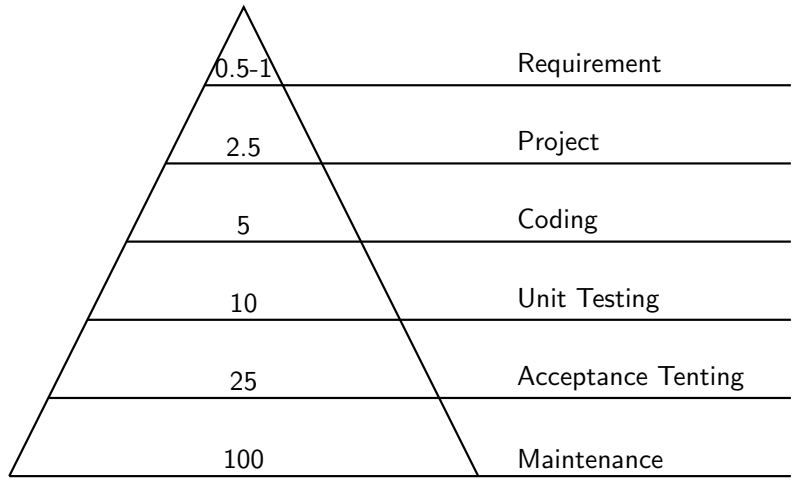
PSSE'2007 – 3rd to 6th December 2007

Why to test? (1)

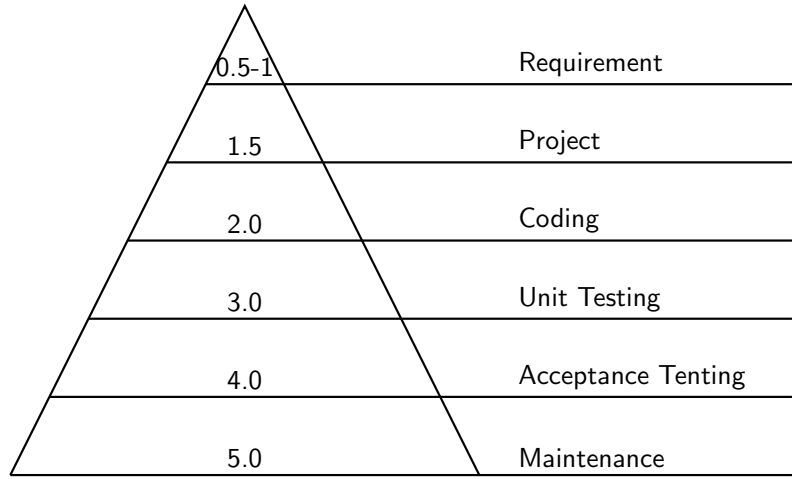
- Increasing interest and importance of software testing, mainly due to the crescent demand for higher software quality.
- Shull et al. (2002) alert that almost no modules are defect-free during development, and after released about 40% of the modules may be defect-free.
- Boehm and Basili (2001) also point out that it is almost improbable to deliver a software product free of defects.
- Moreover, later a fault is detected the greater the cost for its correction.

Why to test? (2)

Cost-escalation factor of defect correction.



Boehm (1987)



Boehm and Basili (2001)

Identifier Program

Test case set based on previous background.

- Program specification (extracted from (Maldonado et al., 2004)):

  The program determines if a given identifier is valid or not in a variant of Pascal language, called Silly Pascal. A valid identifier must begin with a letter and contain only letters or digits. Moreover, it has at least one and no more than six character length.

  Samples of identifiers:

  abc12 (valid);
  cont*1 (invalid); 1soma (invalid); a123456 (invalid)

- We provided a Java and a C version of this specification with a known set of faults for teaching purpose.

In focus

Discuss the theory and practice of software testing focusing on the following testing criteria applied at unit level:

- Functional testing technique
  - Equivalence Partition.
- Structural testing technique
  - Control-flow based criteria: All-Nodes (All-Statements), and All-Edges (Decision Coverage).
  - Data-flow based criteria: All-Uses, and All-Pot-Uses (Maldonado, 1991).
- Fault-Based testing technique
  - Mutation Testing (DeMillo et al., 1978).

What is test? (1)

- According to the IEEE Standard Glossary 610.12-1990 (R2002) (IEEE, 2002):
  "The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component."
- Craig and Jaskiel (2002) present another definition:
  "Testing is a concurrent life cycle process of engineering, using and maintaining testware in order to measure and improve the quality of the software being tested."

What is test? (2)

- Different organizations and people have different testing purposes.
- Software Process *versus* Testing Process.
- Testing Maturity Levels (Beizer, 1990):
  - Level 0 - There's no difference between testing and debugging;
  - Level 1 - The purpose of testing is to show that software works;
  - Level 2 - The purpose of testing is to show that the software doesn't work;
  - Level 3 - The purpose of testing is not to prove anything, but to reduce the perceived risk of not working to an acceptable value;
  - Level 4 - Testing is not an act. It is a mental discipline that results in low-risk software without much testing effort.

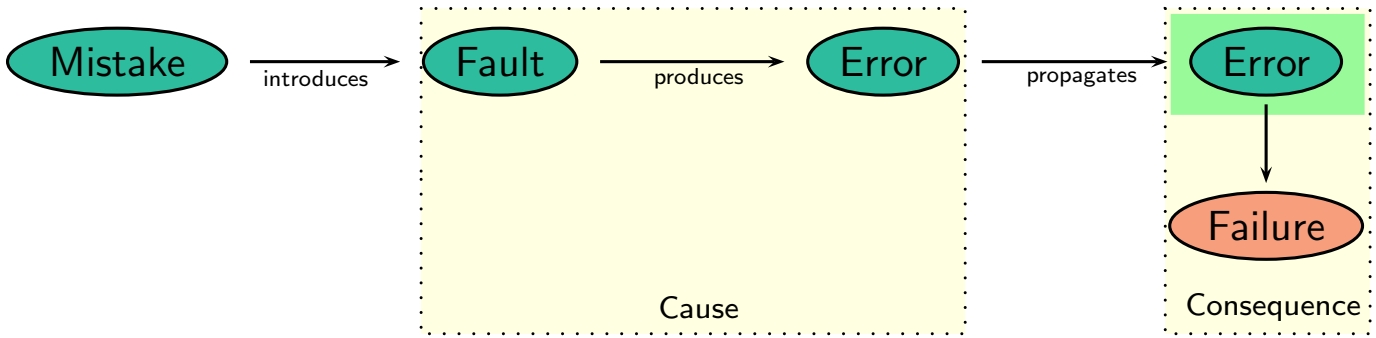
Terminology (1)

IEEE Standard Glossary 610.12-1990 (R2002) (IEEE, 2002) differentiates the terms:

1. **Mistake** – a human action that produces an incorrect result. Example: an incorrect action took by the programmer.
2. **Fault** – an incorrect step, process, or data definition in a computer program. In common usage, the term "error", "bug", and "defect" are used to express this meaning. Example: an incorrect instruction or statement.
3. **Error** – the difference between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition.
4. **Failure** – the inability of a system or component to perform its required functions within specified performance requirements.

Terminology (2)

Standard terminology relation.



Terminology (3)

One considers the following statement: $(z = y + x)$

1. **Mistake** – statement is changed by $(z = y - x)$, characterizing a fault.
2. **Fault** – if activated (executed) by an $x = 0$, no incorrect output is produced. For $x \neq 0$, the fault activation cause an error on variable z.
3. **Error** – the erroneous program state, when propagated to the output, will cause a **Failure**.

Challenges

- Who have already tested a software product?
- What were the biggest challenges?
- Some common problems are:
  - There is not enough time to test properly;
  - There are too many combinations of inputs to test;
  - There is difficulty in determining the expected results of each test;
  - Nonexistent or rapidly changing requirements;
  - There is no training in testing processes;
  - There is no tool support;
  - Management that either does not understand testing or (apparently) does not take care about quality;

Undecidable Problems (1)

Correctness: There is no general purpose algorithm to prove the correctness of a product;

Equivalence: Given two programs, whether they are equivalent; or given two paths (sequence of statements) whether they compute the same function;

Executability: Given a path (sequence of statements) whether there exist an input data that can execute such a path.

Coincident Correction: A product can present, coincidently, a correct result for a given input data $d \in D$ because one fault masks the error of another one.

Undecidable Problems (1)

Correctness: There is no general purpose algorithm to prove the correctness of a product;

Equivalence: Given two programs, whether they are equivalent; or given two paths (sequence of statements) whether they compute the same function;

Executability: Given a path (sequence of statements) whether there exist an input data that can execute such a path.

Coincident Correction: A product can present, coincidently, a correct result for a given input data $d \in D$ because one fault masks the error of another one.

These limitations bring important considerations in the context of software testing, mainly the impossibility of providing a full automation of all necessary testing activities.

Undecidable Problems (2)

Example of Equivalence:

```
1  public class Factorial {
2    public static long compute(int x)
3                    throws NegativeNumberException {
4      if (x >= 0) {
5        long r = 1;
6        for ( int k = 2 ; k <= x; k++) {
7          r *= k;
8        }
9        return r;
10     } else {
11       throw new NegativeNumberException();
12     }
13   }
14 }
```

```
1  public class Factorial {
2    public static long compute(int x)
3                    throws NegativeNumberException {
4      if (x >= 0) {
5        long r = 1;
6        for ( int k = 1 ; k <= x; k++) { //int k = 2;
7          r *= k;
8        }
9        return r;
10     } else {
11       throw new NegativeNumberException();
12     }
13   }
14 }
```

Undecidable Problems (3)

Example of Infeasiability:

```
4  public boolean validateIdentifier(String s) {
5     char achar;
6     boolean valid_id = false;
7     achar = s.charAt(0);
8     valid_id = valid_s(achar);
9     if (s.length() > 1) {
10       achar = s.charAt(1);
11       int i = 1;
12       while (i < s.length() - 1) {
13          achar = s.charAt(i);
14          if (!valid_f(achar))
15             valid_id = false;
16          i++;
17       }
18    }
19
20    if (valid_id && (s.length() >= 1) && (s.length() < 6))
21       return true;
22    else
23       return false;
24 }
```

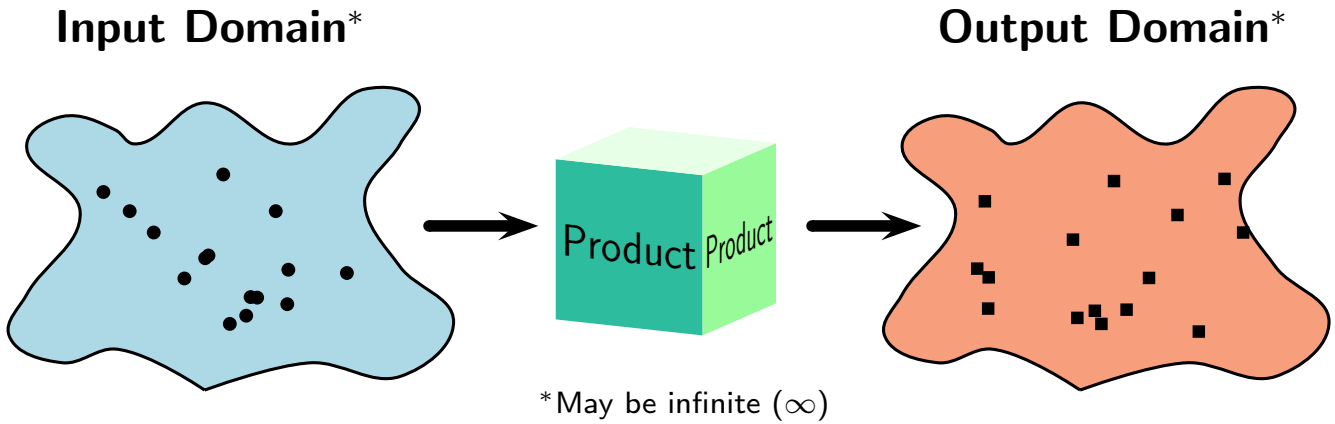
- There is no executable paths that includes the statements in lines 15 and 21.

The Impossibility of Testing Everything (1)

Why do we not perform an exhaustive test?

The Impossibility of Testing Everything (1)

Why do we not perform an exhaustive test?



The Impossibility of Testing Everything (2)

Observe the example below, adapted from Binder (1999):

```
1  int blech(int j) {
2      j = j - 1;  // should be j = j + 1
3      j = j / 30000;
4      return j;
5  }
```

- Considering an integer type of 16 bits (2 bytes) – the lowest possible input value is -32,768 and the highest is 32,767. Thus there are 65,536 possible inputs into this small program.
- Will you have the time to create 65,536 test cases? And for larger programs: How many test cases will be necessary?

The Impossibility of Testing Everything (2)

```
1  int blech(int j) {
2      j = j - 1;  // should be j = j + 1
3      j = j / 30000;
4      return j;
5  }
```

So which input values do we choose?

Input(j)	Expected Output	Actual Result
1	0	0
42	0	0
40000	1	1
-64000	-2	-2

- Note that none of the test cases chosen have detected this defect.
- Which input values will detect the defect?

The Impossibility of Testing Everything (3)

- Only four of the possible 65,536 input values will find this fault:

The Impossibility of Testing Everything (3)

- Only four of the possible 65,536 input values will find this fault:
- These are the input values:

Input(j)	Expected Output	Actual Result
-30000	0	-1
-29999	0	-1
30000	1	0
29999	1	0

The Impossibility of Testing Everything (3)

- Only four of the possible 65,536 input values will find this fault:
- These are the input values:

Input(j)	Expected Output	Actual Result
-30000	0	-1
-29999	0	-1
30000	1	0
29999	1	0

- What is the chance you will choose one of the four???

Design of Test Case

- To be most effective and efficient, test cases must be designed, not just slapped together;
- According to Pressman (2005):
  "The design of tests for software and other engineering products can be as challenging as the initial design of the product itself. Yet ... software engineers often treat testing as an afterthought, developing test cases that 'feel right' but have little assurance of being complete. Recalling the objectives of testing, we must design tests that have the highest likelihood of finding the most errors with a minimum amount of time and effort."

Parts of a Test Case

- Well designed test cases are composed by two mandatory parts and others optional:
  - Pre-conditions;
  - Inputs;
  - Outputs;
  - Order of execution.
- $(d, S(d))$ is a test case, where $d \in D$ is the input and $S(d)$ represents the expected output of d according to the specification S.

Parts of a Test Case – Inputs

- Inputs are commonly thought of as data entered at a keyboard.
- Although, data can come from other sources-data, such as:
  - Data from interfacing systems;
  - Data from interfacing devices;
  - Data read from files or databases;
  - The state the system is in when the data arrives; and
  - The environment within which the system executes.

Parts of a Test Case – Outputs

- Outputs have similar variety as inputs.
- Often outputs are thought of as just the data displayed on a computer screen.
- In addition:
  - Data can be sent to interfacing systems and to external devices;
  - Data can be written to files or databases; and
  - The state or the environment may be modified by the system's execution.

Parts of a Test Case – Oracle (1)

- All of these relevant inputs and outputs are important components of a test case.
- In test case design, determining the expected outputs is the function of an "oracle".
- An oracle is any program, process, or data that provides the test designer with the expected result of a test.

Parts of a Test Case – Oracle (2)

- Beizer (1990) lists five types of oracles:
  - **Kiddie Oracles** – Just run the program and see what comes out. If it looks about right, it must be right.
  - **Regression Test** Suites – Run the program and compare the output to the results of the same tests run against a previous version of the program.
  - **Validated Data** – Run the program and compare the results against a standard such as a table, formula, or other accepted definition of valid output.
  - **Purchased Test** Suites – Run the program against a standardized test suite that has been previously created and validated. Programs like compilers, Web browsers, and SQL (Structured Query Language) processors are often tested against such suites.
  - **Existing Program** – Run the program and compare the output to another version of the program.

Parts of a Test Case – Order of Execution (1)

- There are two styles of test case design regarding order of test execution:
  - Cascading test cases; or
  - Independent test cases.

Parts of a Test Case – Order of Execution (2)

- **Cascading test cases** - Test cases may build on each other.
- For example, the first test case exercises a particular feature of the software and then leaves the system in a state such that the second test case can be executed.
- In testing a database consider these test cases:
  1. Create a record;
  2. Read the record;
  3. Update the record;
  4. Read the record;
  5. Delete the record;
  6. Read the deleted record.
- The **advantage** of this approach is that each test case is typically small and simple.
- The **disadvantage** is that if one test fails, the subsequent tests may be invalid.

Auri M. R. Vincenzi, Márcio E. Delamaro, Erika N. Höhn and José C. Maldonado

Parts of a Test Case – Order of Execution (3)

- **Independent test cases** - Each test case is entirely self contained.
- Tests do not build on each other or require that other tests have been successfully execute.
- The **advantage** is that any number of tests can be executed in any order.
- The **disadvantage** is that each test tends to be larger and more complex and thus more difficult to design, create, and maintain.

Set of test cases

- The term **test set** is used to define a set of test cases.
- When a test set T satisfies all the requirements of a given criterion C, T is said C-adequate.
- From a given T C-adequate it is possible to obtain, in theory, infinite C-adequate test sets by including more test cases in T.
- Test set minimization.

Types Of Testing (1)

- Different types of testing criteria can be used to verify the behavior of a product against its specification.
- Testing is often divided into functional (black box) testing, structural (white box) testing and fault-based testing.
- We refer to these **types of testing** as **testing techniques**.

Types Of Testing (2)

- A testing technique is defined according to the source of information used to carried out the test activity.
  - **Functional or Black box testing** is a technique in which testing is based solely on the requirements and specifications. It requires no knowledge of the internal paths, structure, or implementation of the software under test
  - **Structural or White box testing** is a technique in which testing is based on the internal paths, structure, and implementation of the software under test. It requires detailed programming skills.
  - **Fault-based testing** is a technique in which testing is based on historical information about common faults detected during the software development life cycle.

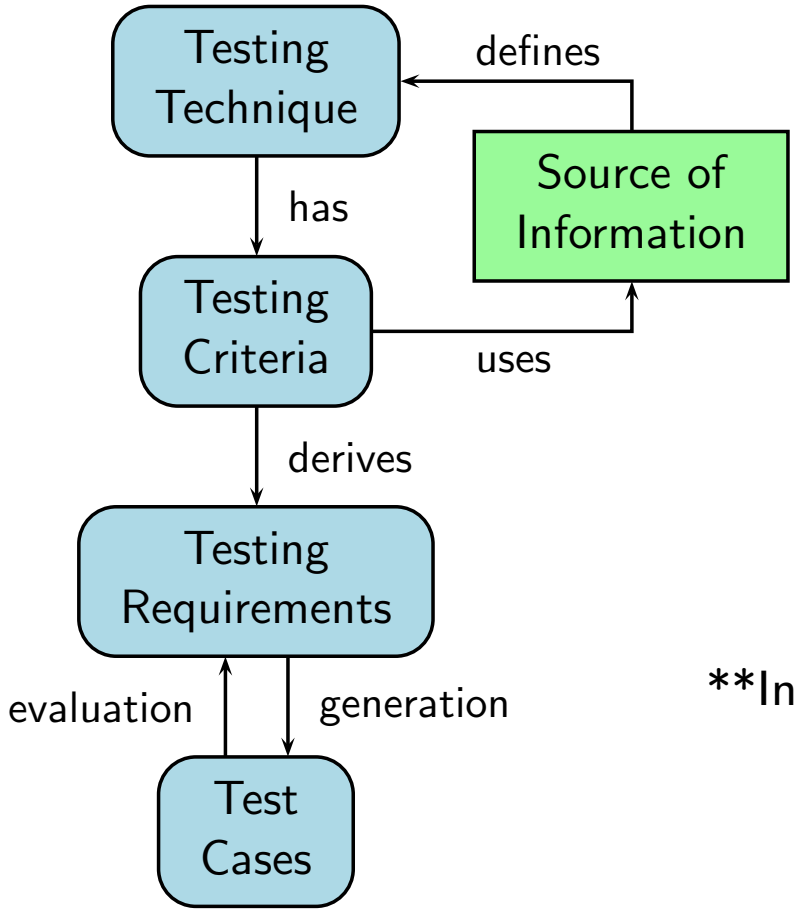
Auri M. R. Vincenzi, Márcio E. Delamaro, Erika N. Höhn and José C. Maldonado

Types Of Testing (3)

- Each testing technique has a set of **testing criteria**.
- A testing criterion systematizes the way **testing requirements** are generated from the source of information (specification, source code, historical fault database, etc.)
- Testing requirements are useful to:
  - Test case generation; or
  - Evaluate the quality of an existent test set.

Auri M. R. Vincenzi, Márcio E. Delamaro, Erika N. Höhn and José C. Maldonado

Testing Techniques, Criteria, and Requirements



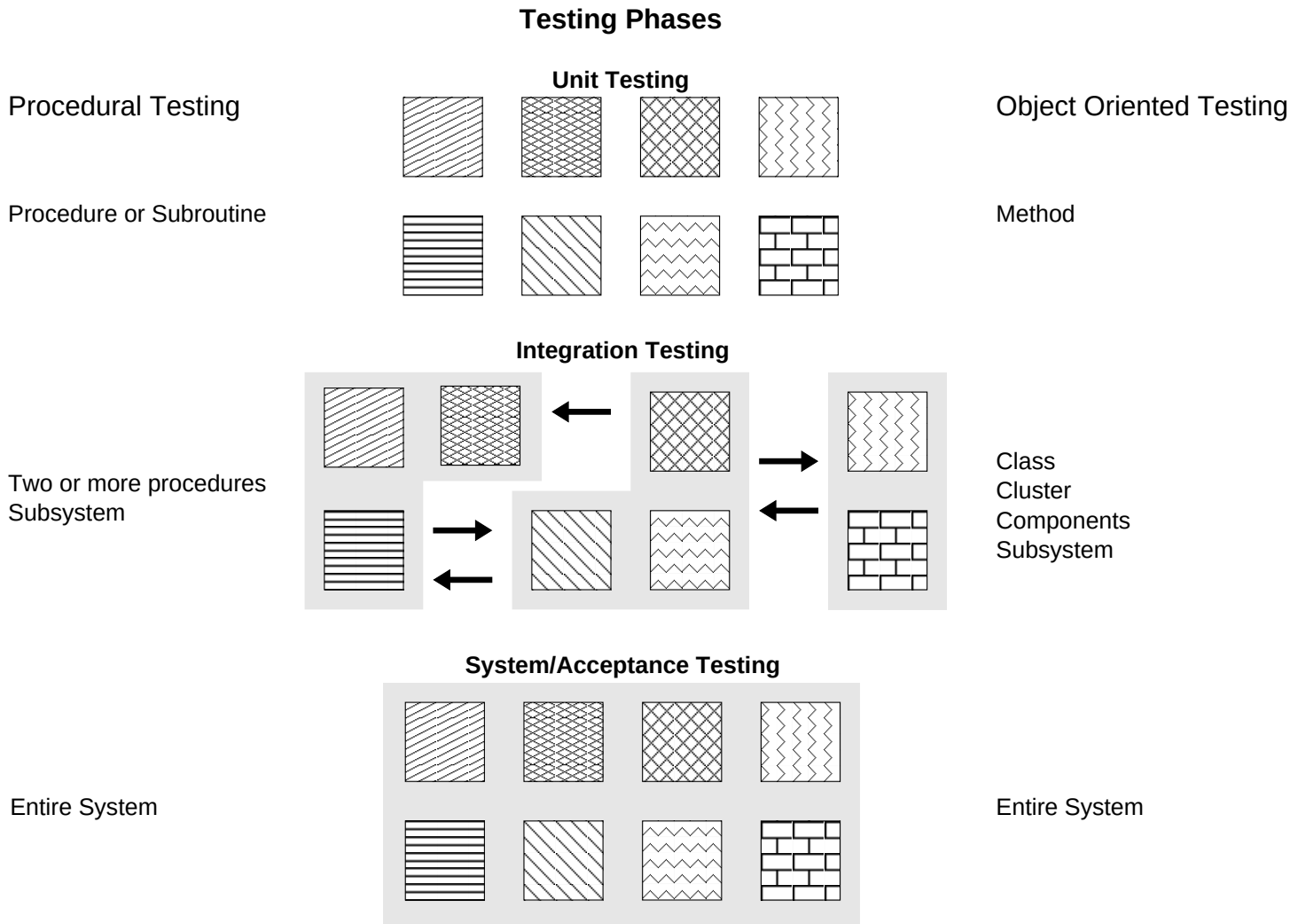
**Infeasible requirements.

Auri M. R. Vincenzi, Márcio E. Delamaro, Erika N. Höhn and José C. Maldonado

Testing Phases (1)

- In the same way as other Software Engineering activities, the Testing Activity is also divided in phases.
- The objective is to reduce its complexity.
- Concept of "divide and conquer".
- One begins testing the minor executable unit and ends testing the complete system.

Testing Phases (2)



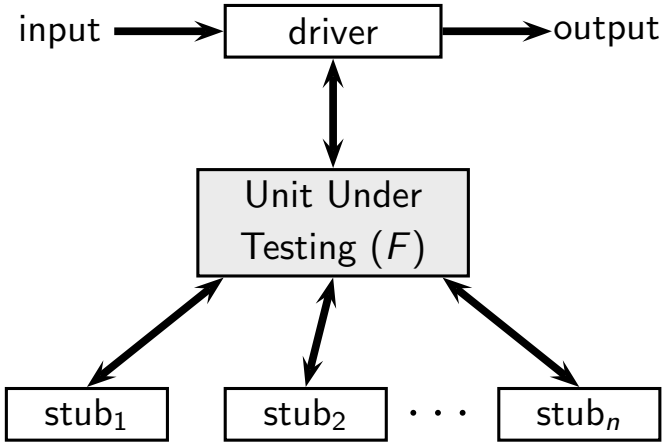
Main testing phases (adapted from Binder (1999)).

Unit Testing

- Aims at identifying programming and logical faults in the program unit.
- Different languages have different units:
  - Pascal and C have procedures and functions.
  - Java and C++ have methods (or classes?).
  - Basic and COBOL (what is considered an unit?).
- How to test an unit which depends on another one to be executed?
- How to test an unit which requires input data provided by another unit?

Driver and *Stub*

- *Drivers* and *stubs* are useful to unit testing.
- The **driver** is responsible to provide the necessary test data to the unit and, later, to present the generated output to the tester.
- The *stub* simulates the behavior of another unit not yet implemented but called by the unit under test.



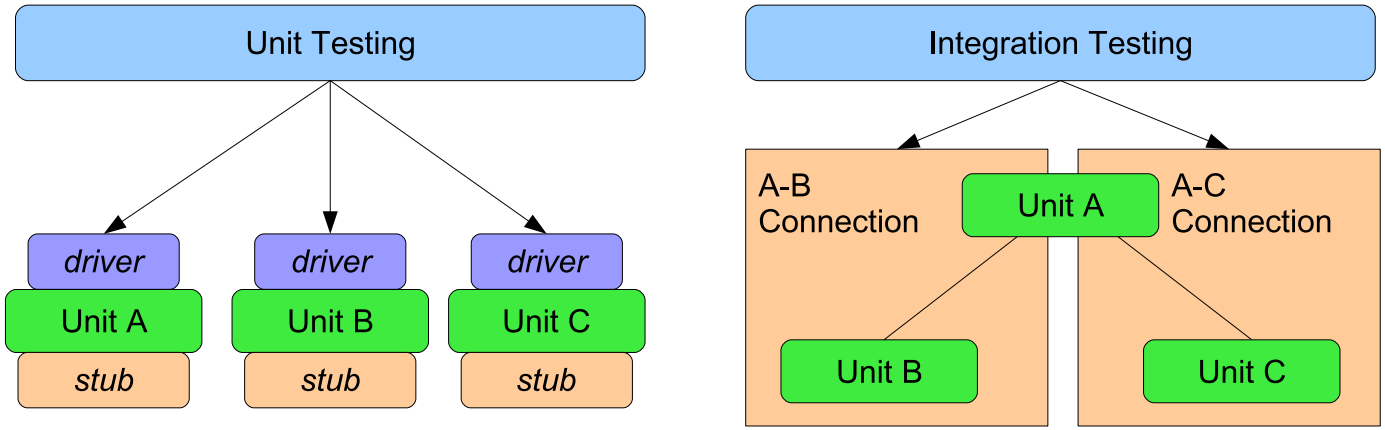
Drivers and *Stubs*.

Integration Testing (1)

- The main objective is to verify if the units tested individually communicate accordingly when integrated together.
  1. Why to perform integration testing if each unit, in isolated, works correctly?

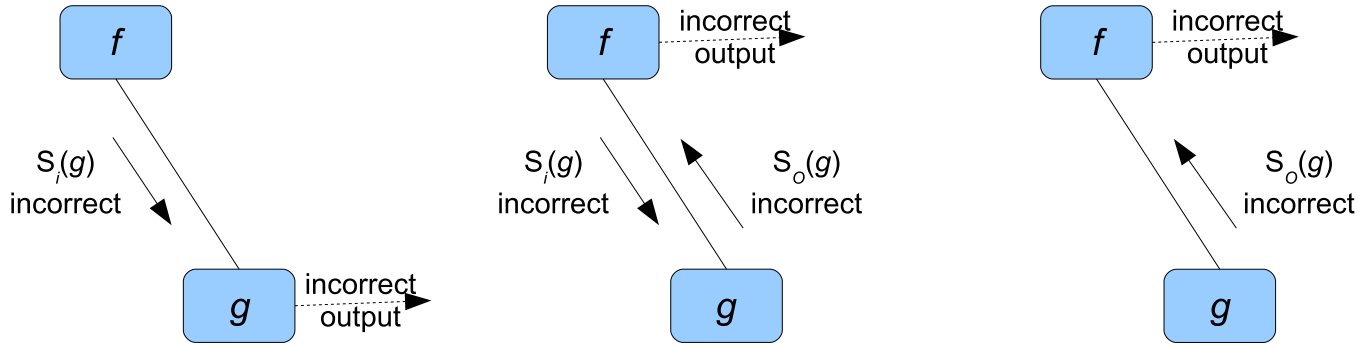
Integration Testing (2)

- Data can be lost in the unit's interface.
- Global variables can suffer undesirable interferences.



Unit X Integration Testing.

Integration Testing (3)



Types of Integration Errors.

System Testing

- A system consists of all units of the software (and possibly hardware, user manuals, training materials, etc.) that make up the product delivered to the customer.
- System testing focuses on defects that arise at this highest level of integration.
- Typically system testing includes many types of testing: functionality, usability, security and localization, reliability and availability, capacity, performance, backup and recovery, portability, and many more. (ISO-IEC-9126 Standard for more information (ISO/IEC, 1991))

Auri M. R. Vincenzi, Márcio E. Delamaro, Erika N. Höhn and José C. Maldonado

Acceptance Testing

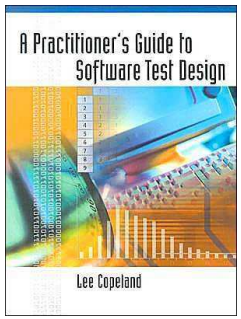
- Acceptance testing is defined as that testing, which when completed successfully, will result in the customer accepting the software.

Summary

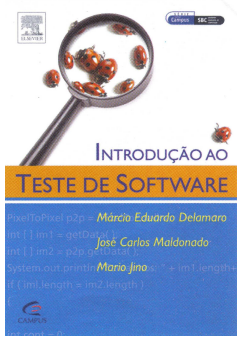
- The testing activity is a process executed in parallel with the program development life cycle.
- The design of tests for software and other engineering products can be as challenging as the initial design of the product itself.
  - The design of test case should consider the main objective of testing: to expose the faults in the program under test.
  - The secret is to discover test cases with higher probability to make the program to produce incorrect outputs.
- The testing techniques and criteria provide support to systematize the test case generation.
- Moreover, to reduce the complexity of the testing activity it is divided in phases.

Recommended Reading

Interested readers can take a look at the first chapters of:



A Practitioner's Guide to Software Test Design
Lee Copeland
Artech House Publishers
Norwood, MA, 2004.



Introdução ao Teste de Software
Márcio Eduardo Delamaro
José Carlos Maldonado
Mario Jino
Campus Elsevier
Rio de Janeiro, RJ, 2007.

References I

B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Company, New York, 2nd edition, 1990.

R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*, volume 1. Addison Wesley Longman, Inc., 1999.

B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.962984.

B. W. Boehm. Industrial software metrics top 10 list. *IEEE Software*, 4(5):84–85, September 1987.

R. D. Craig and S. P. Jaskiel. *Systematic Software Testing*. Artech House Publishers, 2002.

R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–43, April 1978.

IEEE. IEEE standard glossary of software engineering terminology. Standard 610.12-1990 (R2002), IEEE Computer Society Press, 2002.

ISO/IEC. Quality characteristics and guidelines for their use. Padrão ISO/IEC 9126, ISO/IEC, December 1991.

J. C. Maldonado. *Potential-Uses Criteria: A Contribution to the Structural Testing of Software*. PhD thesis, DCA/FEE/UNICAMP, Campinas, SP, Brazil, July 1991. (in Portuguese).

J. C. Maldonado, E. F. Barbosa, A. M. R. Vincenzi, M. E. Delamaro, S. R. S. Souza, and M. Jino. Introdução ao teste de software. Technical Report 65 – Version 2004-01, Instituto de Ciências Matemáticas e de Computação – ICMC-USP, April 2004. Available at: http://www.icmc.usp.br/~biblio/index.php?destino=notas_didaticas. Accessed on: 02/19/2007 (in Portuguese).

R. S. Pressman. *Software Engineering – A Practitioner's Approach*. McGraw-Hill, 6 edition, 2005.

F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R.e Tesoriero, and M. Zelkowitz. What we have learned about fighting defects. In *VIII International Symposium on Software Metrics - METRICS'02*, pages 249–258, Washington, DC, USA, June 2002. IEEE Computer Society. ISBN 0-7695-1339-5.