Testing

Fondamenti teorici

Riferimenti

- Ian Sommerville, Ingegneria del Software, capitoli 22-23-24 (più dettagliato sui processi)
- Pressman, Principi di Ingegneria del Software, 5° edizione, Capitoli 15-16
- Ghezzi, Jazazeri, Mandrioli, Ingegneria del Software, 2° edizione, Capitolo 6 (più dettagliato sulle tecniche)

IEEE definitions for testing

IEEE Std.729-1983

- (2) The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item.

- See also: acceptance testing; benchmark; checkout; component testing; development testing; dynamic analysis; formal testing; functional testing; informal testing; integration testing; interface testing; loopback testing; mutation testing; operational testing; performance testing; qualification testing; regression testing; stress testing; structural testing; system testing; unit testing.

Standard IEEE 610.12-1990 :

- (1) 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.

IEEE definitions for testing

Standard IEEE 829-2008:

The testing process provides objective evidence that the software-based system and its associated products:

a) Satisfy the allocated system requirements

b) Solve the right problem (e.g., correctly model physical laws, implement business rules, and use the proper system assumptions)

c) Satisfy its intended use and user needs

testing: (A) An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.
(B) To conduct an activity as in (A).

IEEE definitions for testing

IEEE 29119 - 2013 (only partially released)

Testing: set of activities conducted to facilitate discovery and/or evaluation of properties of one or more test items.

Testing activities include planning, preparation, execution, reporting, and management activities, insofar as they are directed towards testing.

test item: work product that is an object of testing

EXAMPLES: A system, a software item, a requirements document, a design specification, a user guide.

Verifica e Validazione del Software

- L'attività di Verifica e Validazione (V & V) punta a mostrare che il software è conforme alle sue specifiche (Verifica) e soddisfa le aspettative del cliente (Validazione).
  - *Verifica: Are we making the product right?*
  - *Validazione: Are we making the right product?*
- Due approcci complementari alla verifica:
  - Approccio **sperimentale** (test, o analisi dinamica, del prodotto)
  - Approccio **analitico** (analisi statica del prodotto e della sua documentazione, verifica formale)

Definizioni

- **Errore** (umano)
  - incomprensione umana nel tentativo di comprendere o risolvere un problema, o nell'uso di strumenti
- **Difetto** (fault o bug)
  - Manifestazione nel software di un errore umano, e causa del fallimento del sistema nell'eseguire la funzione richiesta
- **Malfunzionamento** (failure)
  - incapacità del software di comportarsi secondo le aspettative o le specifiche
  - un malfunzionamento ha una natura dinamica: accade in un certo istante di tempo e può essere osservato solo mediante esecuzione

Un esempio

```
void raddoppia()
{
cin>>x;
y := x*x;
cout<<y;
}
```

ERRORE di editing/digitazione

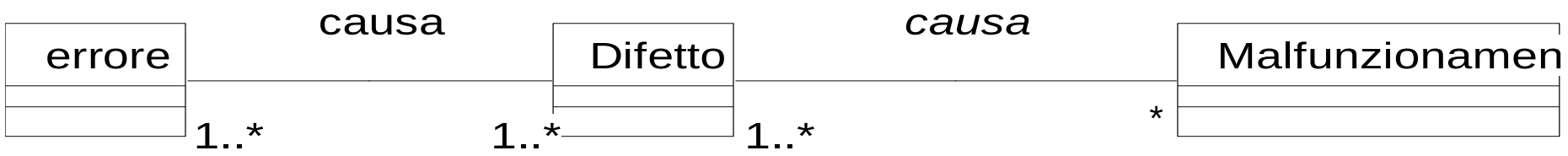
ERRORE
siamo convinti che il raddoppio si calcoli come x*x

DIFETTO
“*” invece di “+”

MALFUNZIONAMENTO
=> il valore visualizzato è errato

Possibile MALFUNZIONAMENTO in esecuzione…
(può verificarsi o meno: dipende dall’input)

Relazione fra Errore, Difetto e Malfunzionamento



9/9

0800 Arctan started
1000 " stopped - arctan ✓ { 1.2700 9.037 847 025
9.037 846 795 correct
13"uc (032) MP - MC ~~1.9821647000~~ ~~2.13047641 5 (-3)~~ 4.615925059(-2)
(033) PRO 2 2.13047641 5
correct 2.13067641 5

Relays 6-2 in 033 failed special speed test
In relay " 11.00 test.
Relays changed

1100 Started Cosine Tape (Sine check)
1525 Started Multy Adder Test.

1545 
Relay #70 Panel F
(moth) in relay.

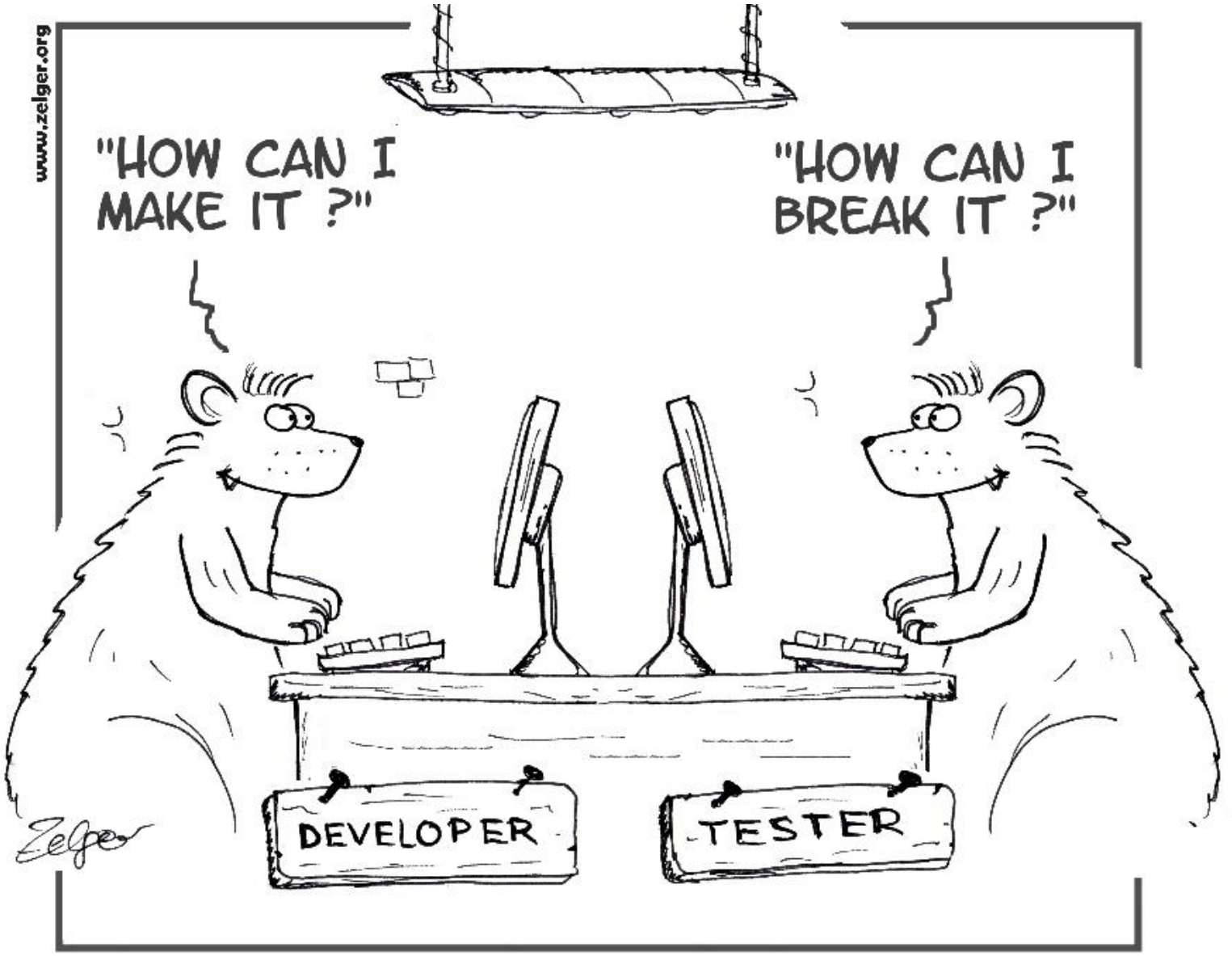
First actual case of bug being found.

1630 Arctangent started.
1700 closed down.

Test dei Difetti

- Ha lo scopo di scoprire i difetti di un programma.
- Approccio usato: scoprire la presenza di difetti osservando i malfunzionamenti!
- Un test dei difetti *ha successo* se porta il programma a comportarsi in maniera scorretta (cioè esibisce malfunzionamenti)

- Il testing può dimostrare la presenza dei difetti.
  - Potrà dimostrare anche la loro assenza?



They weren't so much different, but they had different goals

Ulteriori Definizioni

- Test case: caso di prova di un software
  - Corrispondente all'esecuzione del software con una combinazione d dei propri valori in ingresso

- Test suite: insieme di casi di prova di un software

Storia del testing

- Tracce di testing nella preistoria della IS
  - con la prima linea di codice e' nato anche il testing ...
  - I primi articoli sul testing risalgono al 1949
- Negli anni 70 i primi seri tentativi di fornire dei fondamenti teorici ed approcci sistematici
  - il sogno del testing esaustivo ...
    - *Le relative impraticabili definizioni ad esempio:*
      - **l'obiettivo del testing e' quello di dimostrare la correttezza dei programmi**
      - **un test perfetto si ha quando il programma non contiene errori**
  - Goodenhough and Gerhard "Toward a theory of test data selection" IEEE TSE SE-1(2) pp.156-173, 1975

Storia del testing

- Gli anni 80 eliminano i sogni, consolidano un patrimonio di riferimento; i risultati acquisiti aprono ad una visione: il testing come processo complesso dai costi estremamente elevati.

- **il testing e' il processo di esecuzione di un programma con l'obiettivo di trovare gli errori**
  - Un test che non rivela errori e' un test fallito

- la progettazione, pianificazione ed implementazione del processo di testing inizia con i primi passi di un qualsiasi processo di produzione, manutenzione, evoluzione del software

Storia del testing

- Anni ’90
  - il testing e' l'analisi e l'esecuzione sotto condizioni controllate dei vari componenti di un progetto software con l'obiettivo di mettere in evidenza difetti di qualità
  - Tra le molte sfide :
    - *come abbattere i costi di test (40-60% del costo totale di produzione)*
    - *come approcciare il testing di sistemi object oriented*
    - *come risolvere il testing in manutenzione ed evoluzione*
    - *come aumentare i livelli di automazione dei processi di testing*

- Rilevante esigenza di approcci ingegneristici, quantitativi, di esperimenti controllati di feedback da esperienze ed applicazioni in ambienti reali: tutto ciò costituisce anche il presupposto per nuovi significativi avanzamenti nella composizione del puzzle teorico e modellistico del testing.

Fondamenti teorici del testing

- Consideriamo un programma P come una funzione da un insieme di dati D (dominio) in un insieme di dati R (codominio).
  - Il risultato P(d) ottenuto eseguendo P sul dato di ingresso d, con $d \in D$, è corretto se soddisfa le specifiche, non corretto se diverso dal risultato previsto dalle specifiche.
- La correttezza di un programma P rispetto ad un ingresso d è indicata con ok(P,d)
- Un programma è corretto $ok(P) \Leftrightarrow \forall\ d \in D,\ ok(P,d)$

Adeguatezza dei test

- Scopo dell'attività di testing è la rilevazione di malfunzionamenti.
- Una test suite T rileva un malfunzionamento se il programma P non è corretto per almeno un test case di T.
  - T ha successo per un programma P se rileva uno o più malfunzionamenti presenti in P,
  - viceversa è *inadeguata* una test suite T per la quale esistano dei malfunzionamenti in P che nessun test case è in grado di rilevare
- Una test suite T è detta *ideale* se l'assenza di malfunzionamenti rilevati implica l'assenza di malfunzionamenti
  - *Cioè se* $ok(P,T) \Rightarrow ok(P)$
    - *Ovvero, se qualsiasi test ulteriore non potrà scoprire malfunzionamenti che non siano stati già scoperti da T*

Problemi indecidibili

- Il settore del testing e' tormentato da problemi indecidibili
  - un problema e' detto indecidibile (irrisolvibile) se e‘ possibile dimostrare che non esistono algoritmi che lo risolvono
    - *Problema della terminazione della macchina di turing, teorema della esistenza di funzioni non calcolabili, …*

Stabilire se l'esecuzione di un programma termina a fronte di un input arbitrario e' un problema indecidibile

Altri problemi indecidibili

- Braierd Landerweber 1974
  - dati due programmi, il problema di stabilire se essi calcolano la stessa funzione e' indecidibile

- Enormi conseguenze per il testing:
  - dato un programma e supposto noto e disponibile l'archetipo idealmente corretto di tale programma non possiamo comunque dimostrare l'equivalenza dei due
  - Non esiste un algoritmo in grado di stabilire se due generici cammini del grafo di flusso di controllo di un programma calcolino la stessa funzione o meno (teorema di equivalenza dei cammini)

Teorema di Weyuker

- Dato un generico programma P i seguenti problemi risultano indecidibili:
  - Esiste almeno un dato di ingresso che causa l'esecuzione di un particolare comando?
  - Esiste un particolare dato di ingresso che causa l'esecuzione di una particolare condizione (branch)?
  - Esiste un dato di ingresso che causa l'esecuzione di un particolare cammino?
  - E' possibile trovare almeno un dato di ingresso che causi l'esecuzione di ogni comando di P?
  - E' possibile trovare almeno un dato di ingresso che causi l'esecuzione di ogni condizione (branch) di P?
  - E' possibile trovare almeno un dato di ingresso che causi l'esecuzione di ogni cammino di P?

Tesi di Dijkstra

- **Tesi di Dijkstra**
  - Il testing non può dimostrare l'assenza di difetti, ma solo la loro presenza

- Non vi è garanzia che se alla n-esima prova un modulo od un sistema abbia risposto correttamente (ovvero non sono stati più riscontrati difetti), altrettanto possa fare alla (n+1)-esima

- Impossibilità di produrre tutte le possibili configurazioni di valori di input (test case) in corrispondenza di tutti i possibili stati interni di un sistema software

- **"We did about 10,000 tests on it, and it was working fine until Monday."**
  - Anonymous - Spokesperson for 7-11 after Y2K-related failure of their credit card processing on 2001-01-01

Problemi e Limitazioni

- **La correttezza di un programma è un problema indecidibile!**

- **Problemi:**
  - non vi è garanzia che se alla n-esima prova un modulo od un sistema abbia risposto correttamente (ovvero non sono stati più riscontrati difetti), altrettanto possa fare alla (n+1)-esima
  - Impossibilità di produrre tutte le possibili configurazioni di valori di input (test case) in corrispondenza di tutti i possibili stati interni di un sistema software

Assenza delle proprietà di continuità

- In molti campi dell'ingegneria, il testing è semplificato dall'esistenza di proprietà di **continuità**
  - Se un ponte resiste ad un carico di 1000 tonnellate, allora resisterà anche a carichi più leggeri

- Nel campo del software si ha a che fare con sistemi **discreti**, per i quali piccole variazioni nei valori d'ingresso possono portare a risultati scorretti
  - Il testing esaustivo (ideale) è condizione necessaria per poter valutare la correttezza di un programma a partire dal testing

Exhaustive testing

- Exhaustive testing consists consists of the execution of all the possible behaviours of a software system
  - If exhaustive testing does not show any failure, the program is correct
- Is it always possible to generate and execute the exhaustive test suite?
  - If the program has no branches and have no inputs, then the exhaustive test suite exists and is composed of a single test case
    - *It is possible to test exhaustively the «Hello, World» program*
  - What if the program has branches and the program execution depends on input values?
  - What if the program has loops?

Dependency on input values

```
char x;
cin>>x;
```

In how many ways can be executed this code?

x is a char (with 256 possible values), so the code can be executed in 256 different ways but …

the >> operator allows the insertion of unlimited input sequences:

abcde0

qiunonooiiouuoiounoinoinoinonoiioio0

‘string causing buffer overflow’

0

...

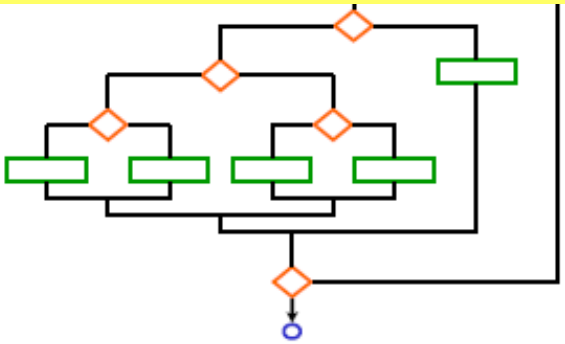
What if the program has branches and loops?

By excluding the loop, the program can be executed following 5 different paths.

By considering the loop, the program can be executed following an unlimited number of paths

By limiting the execution to n loops, the program can be executed in 5^n ways

The problem of the execution of the exhaustive test suite needs an exponential time!



```
.....
repeat
  B0
  if R1 then
        if R2 then
                if R3 then
                        B1
                else
                        B2
                endif
                if R4 then
                        B3
                else
                        B4
                endif
              else
                B5
        endif
  endif
until R6
.....
```

Amenità sul Testing

da "The Zen of Programming" - Geoffrey James

Thus spoke the master: "Any program, no matter how small, contains bugs"

The novice did not believe the master's words. "What if the program were so small that it performed a single function?" he asked.

"Such a program would have no meaning," said the master, "but if such a one existed, the operating system would fail eventually, producing a bug".

But the novice was not satisfied. "What if the operating system did not fail?" he asked.

"There is no operating system that does not fail," said the master, "but if such a one existed, the hardware would fail eventually, producing a bug".

The novice still was not satisfied. "What if the hardware did not fail?" he asked.

The master gave a great sigh. "There is no hardware that does not fail," said the master, "but if such a one existed, the user would want to do something different, and this too is a bug."

A program without bugs would be an absurdity, a nonesuch. If there were a program without any bugs then the world would cease to exist.

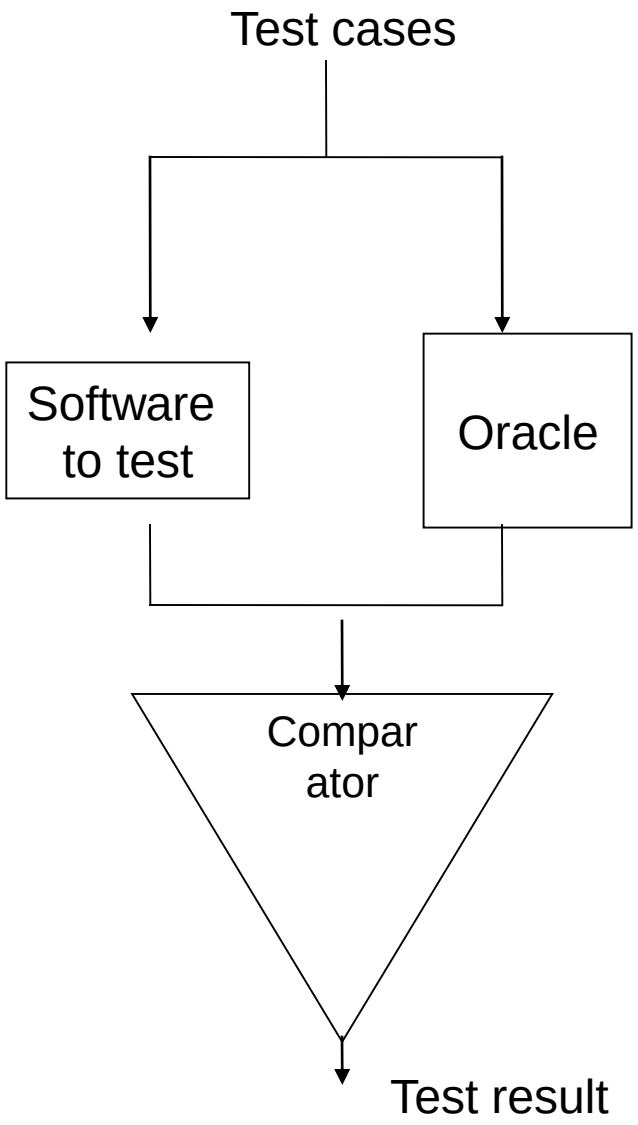
Testing quality requirements

1. Accuracy
2. Repeatability
3. Fault Localization Ability
4. Effectiveness
5. Efficiency

- …

Testing quality requirements: accuracy

- The success of the execution of a test case should be evaluated in an objective way
- In order to design a test case, the **Oracle** has to know exactly what is the expected behavior of the system under test in response to the test case
- The **Oracle** knows the expected behavior of the software system for each test case
- The **Comparator** is able to compare the oracle expected behavior and the tested software behavior and to evaluate if a failure has occurred



Who is the Oracle?

- Human oracle
  - He evaluates the success of the test cases on the basis of the requirements and of his personal judgement
    - *For example, in acceptance testing*
- Software oracle
  - An Oracle is another software having the same behavior of the software to be tested
    - *For example, a known version of bubblesort can be used to test a faster quicksort*
    - *A previous version of a software can be the oracle to evaluate the behavior of a newer software offering the same functionality (regression testing)*
- Implicit oracle
  - Testing against crashes, the oracle found a failure in each case a crash occurs
- Formal specification oracle
  - If the requirements are expressed in a formal way, then the oracle can be automatically derived from them

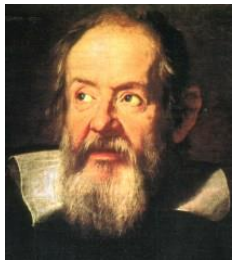
What is the role of the comparator?

- If the oracle provides exact expected values for the output, the comparator has to evaluate a simple bit comparison
- If the oracle is expressed in terms of a set of valid values, the comparator has to evaluate if the obtained result belongs to this set
  - For example, the expected behavior can be expressed as «a positive number», then the comparator has to evaluate a «greater than» condition
- If the oracle is expressed in terms that are not exactly represented by the computer, then the comparator has to perform an approximate evaluation
  - If the expected behavior is the number PI, then the comparator has to compare the difference between the obtained result and an approximate representation of PI
  - If the expected behavior is an image of Naples, the comparator has to compare the output image with a base of Naples images in order to argue if the image is sufficiently similar to a Naples image

Testing quality requirements: accuracy

- Testing should be accurate
  - The success of the execution of a test case should be evaluated in an objective way
    - Executions involving real numbers may produce an approximate result
    - The set of variables that have to be evaluated could be very large
      - For example, the output of a test could be an image that have to be interpreted
    - The output of the test could be subject to human judgement
    - The functional requirements could be expressed in an ambiguous way

Testing quality requirements: repeatability



- Test cases should be repeatable
  - Any experiment must be reproducible under controlled conditions
    - *Are we always able to control any part of the execution environment?*
      - **We could use a virtual machine to have a controlled environment**
    - *Are we always able to restore the state of the execution environment to its conditions before the execution of the tests?*
      - **We could save and restore a snapshot of a virtual machine before and after each test execution**
    - *Does the program execution depend on random values?*
      - **We could use pseudo-random generators and set a seed before starting each test**
    - *Does the program execution depend on races?*
      - **We could force specific sequence of execution by using time and synchronization functions**

Testing quality requirements: fault localization

- Testing should help in fault localization
  - To reduce the effort of the following debugging activities
    - Many simple test cases are better than few complex test case in fault localization
    - A complex test case has a greater probability to fail due to more than one fault

Testing Quality Requirements: effectiveness

- The (main) objective of the testing activities is to find the largest number of faults
  - A test suite T1 is more effective than another test suite T2 if it is able to find more different faults
    - *But it is not implied that if T1 has more test cases founding faults, it will find more different faults*
    - *And it is not implied that if T1 finds more failures, it will find more (different) faults*
    - *An obvious strategy to increase effectiveness is to execute more and more test cases …*

Testing quality requirements: efficiency

- Any testing activity has an associated effort
  - Effort can be expressed in terms of the cost of the execution of the test suite
  - Testing effort can depend on:
    - *Number of test cases;*
    - *Total complexity of the test cases (e.g. number of executed statements)*
    - *Time needed to execute the test cases*
    - *Cost of the resources (human and machines) needed to execute the test cases*
- A test suite T1 is more *efficient* than another test suite T2 if it is able to find the same number of different failures with less effort

Effectiveness vs Efficiency

- The basic strategy of increasing the test suite size may increase the effectiveness but probably reduces the efficiency
- Usually, effectiveness and efficiency are two opposite factors:
  - If there are important time or cost constraints, efficiency is considered more important than effectiveness
  - In critical systems, effectiveness is more important than efficiency
  - In general, the maximization of effectiveness and efficiency is a multi-objective problem
- Often, test suites manually designed by an expert tester are more effective and efficient than test suites automatically generated by a testing tool …
  - But the total effort related to test suite generation and execution may be larger

Effectiveness and efficiency measures

- **Effectiveness** can be measured as:
  Number of different faults found / Number of existing faults
- **Efficiency** can be measured as:
  Number of test cases founding faults / Number of executed test cases
  - … but the Number of existing faults is generally unknown
  - … and the number of executed test cases is not a measure of the effort related to their execution
  - … and test cases founding all the same fault should not contribute positively to efficiency

Effectiveness and efficiency measures

- The proposed measures are not very useful for an absolute evaluation but can be considered for relative measures

- Given two test suites T1 and T2, two possible measures of effectiveness and efficiency

  Relative Effectiveness (T1, T2) = Number of different faults found by T1 / Number of different faults found by T2

  Relative Efficiency (T1, T2) = Number of test cases of T1 founding faults / Number of test cases of T2 founding faults

Approximate measures: code coverage

- The «exact» measures of effectiveness and efficiency depend on the real presence of faults. If a software has no residual bugs, the effectivess and efficiency of each test suite are always zero
  - But we do not know if there are any bugs, so we cannot avoid testing
  - We need approximate measures of effectiveness and efficiency useful in cases there few or no bugs

- An approximate measure of effectiveness is code coverage:
  - CovLOC=Number of LOCs covered by at least a test case / Total number of LOCs
  - CovMethod=Number of methods called by at least a test case / Total number of methods
  - CovClass= Number of classes used by at least a test case /Total number of classes

Code coverage ratio

- If a LOC contains a fault and no tests execute it, then the fault SURELY will not be found
- If a LOC contains a fault and some tests execute it, then it is POSSIBLE that the fault will cause a failure

- In general, more the quantity of code that we execute, more the «hope» to found faults
- If two test suites T1 and T2 will found the same number of faults (possibly 0 faults), and T1 cover more LOCs than T2, we could have a better confidence in test suite T1

Code coverage problems and limitations

- The higher granularity of code coverage is related to executable code, but the translation of source code in executable code depends also on the compiler behavior
  - For example, the statement:

    if (a>b && x>y && f(&x)==5)

    is translated with a complex set of executable statements.
  - Some compilers translate the expression from left to right, some others from right to left. Some compilers go to the else branch just when one of the condition is false, some others evaluate always all the conditions
  - The evaluation of code coverage at the level of source code is usually approximated by considering fractional values of LOC coverage.
- The total amount of LOCs is generally available for compiled code …
  - … but it is surely not available for interpreted code
    - *… what about languages such as Javascript (in particular, AJAX)?*
  - … what about languages such as Java that dynamically transform bytecode in executable code depending on the target machine?

Problema della generazione di codice a run-time

- In molti linguaggi è possibile generare codice a tempo di esecuzione
  - una pagina server può generare codice lato client (anche Javascript eseguibile) diverso in seguito ad ogni diversa esecuzione
  - Una shell «esegue» esattamente ciò che gli viene immesso come input (che sia un eseguibile o uno script batch)
  - L'istruzione eval (o evaluate) disponibile in molti linguaggi di programmazione (specialmente in quelli interpretati) consente di eseguire codice «arbitrario» formato a tempo di esecuzione

- In questi casi non è possibile definire il quantitativo esatto di codice eseguibile da coprire!

Test case specification

- Minimal set of information able to describe a test case specfication
  - ID Number and Description
  - Preconditions
    - *Hypotheses that must be veriified before the test is executed*
  - Input values
  - Expected output values (the «Oracle»)
  - Expected Postconditions
    - *Hypotheses that must be verified after the test is executed*

ID	Precond	Input	Expected Output	PostCond

Test case execution report

- In addition to the information needed for test case specification:
  - Output values
  - Result
    - *Success if preconditions and postconditions are true and output values are judged equivalent to expected output values*
    - *Failure if preconditions and postconditions are true but output values are not judged equivalent to expected output values*
    - *Not applicable (N/A) if al least a precondition or a postcondition is not true*

ID	Precond	Input	Expected Output	Output	PostCond	Result

Input and Output

- Input and output can be described as attribue-value pairs …
- … or by a stream
  - For example, they can be represented by a text file
- Input may be described by a sequence of actions
  - For example in GUI based applications, input may be represented by a sequence of user actions
- Input and output may also be described by a set of signals and by the time of arrival to the system / exit from the system

Preconditions and postconditions

- Preconditions and postconditions may be about:
  - The existence and state of external services or data sources
    - *For example files, databases, services, global variables …*
  - The state of the application before/after the test execution
    - *For example: the correct authentication, the presence or absence of such data in the database, the reaching of a specific interface*
      - ***Usually, if a test is quite complex, then intermediate conditions are added to improve the fault localization ability of the test case***

Testing Termination Criteria

- Due to the Dijkstra theorem, the problem of automatic testing termination is undecidable
- Some common criteria to terminate testing:
  - Time criteria: tests stop after a fixed time
  - Cost/Effort criteria: tests stop when a fixed cost or effort is reached
  - Objective criteria: tests stop when a fixed testing objective is reached
    - *E.g. a fixed number of bugs is found or a fixed percentage of code coverage is reached*
  - Statistical criteria: when the temporal frequency of fault discovery goes under a fixed value
    - *For example, when no new faults have been found in the last set of n test cases*
  - Saturation criteria: when a set of different testing executions give exactly the same results
    - *Applicable, for example, in problems involving randomness*

Software can be chaotic, but we make it work

Expert

Trying Stuff Until it Works

O RLY?

The Practical Developer
@ThePracticalDev

Tipologie fondamentali di testing

- In base all'obiettivo:
  - Testing funzionale
  - Testing strutturale
  - Quality Assessment (Stress Testing, Security Testing, Performance Testing, …)
- In base all'oggetto del testing
  - Testing di sistema
  - Testing di integrazione
  - Testing di unità
- In base alle informazioni disponibili:
  - Testing Black Box
  - Testing Grey Box
  - Testing White Box
- …

Tipologie fondamentali di testing

- Articolo consigliato:
  - **Orso, Rothermel, Software Testing: A Research Travelogue (2000–2014)**
    - http://www.cc.gatech.edu/~orso/papers/orso.rothermel.ICSE2014-FOSE.pdf
- Elenca l'evoluzione temporale delle principali linee di ricerca del testing tra il 2000 e il 2014
  - Risorsa utile per molti dei concetti che verranno visti più in dettaglio negli argomenti successivi