

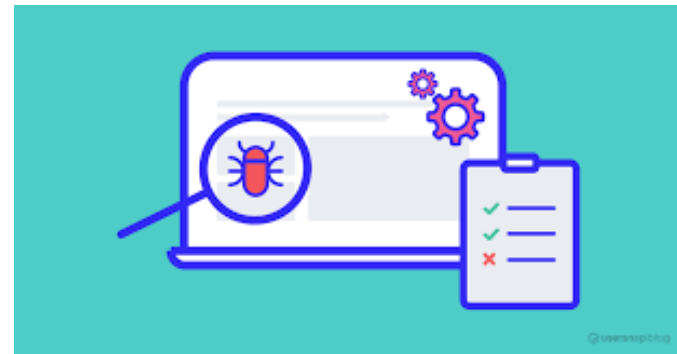
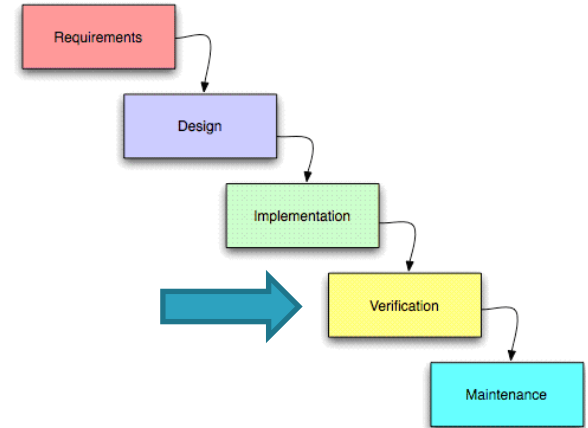


SOFTWARE TESTING: INTRODUZIONE

Ingegneria del software 2023/2024

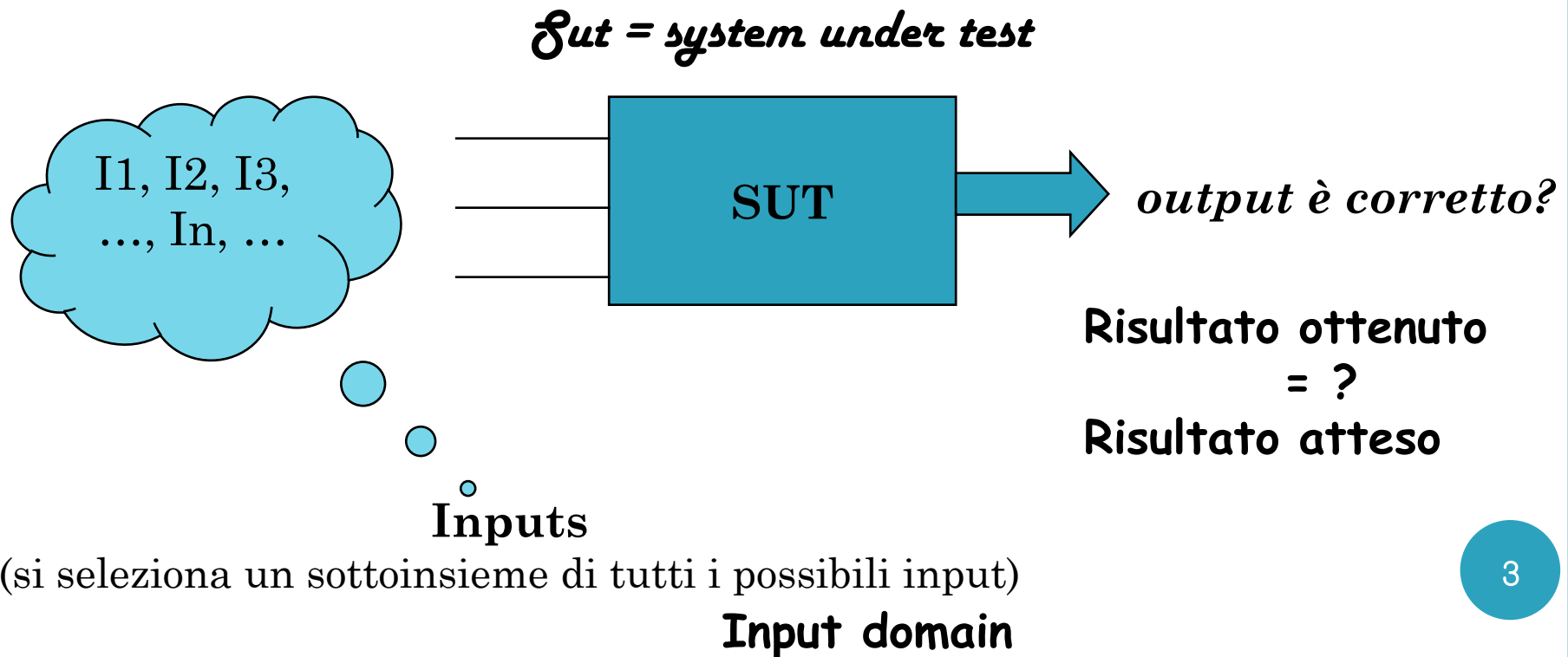
AGENDA

- Cosa è il **Software Testing**?
- Perché è difficile?
- Testing strutturale e funzionale
 - White-box vs. Black box
- Testing di unità, integrazione, sistema e regressione
- **White-box testing**
 - Control flow graph
 - Criteri di copertura
- **Black-box testing**
 - Equivalence partitioning
 - Boundary value analysis



COSA È IL SOFTWARE TESTING?

- E' una **procedura sistematica** che prevede l'**esecuzione** di un sistema software (SUT) con l'intento di trovare **failure** (e poi il **fault** associato)



ERRORE, FAULT E FAILURE

- L'**errore** è commesso da uno sviluppatore
 - Es. funzione 'C/C++' che deve raddoppiare un numero
 - $y = 2 * x$
- L'errore può essere
 - un errore di battitura
 - un errore concettuale
 - non è chiaro il significato di "raddoppiare"
- **Fault**: alla linea 3
 - è stato usato "**x**" invece di "**2**"
- **Failure**: se eseguo reDouble(3) ottengo 9 e non 6 ...

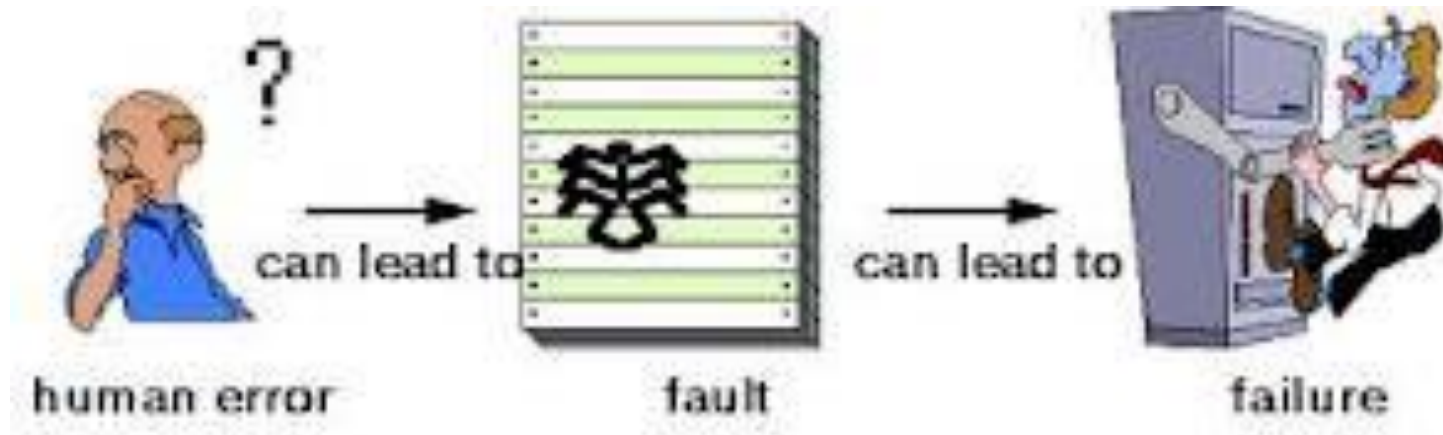
Funzione 'C/C++' che deve raddoppiare un numero

Codice

```
1  int reDouble (int x);  
2  {  int y;  
3    y := x * x;  
4    return (y);  
5  }
```

2

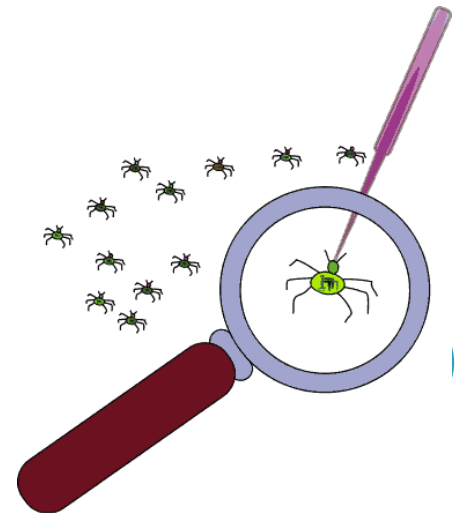
CATENA CAUSALE



Attenzione: un fault può sopravvivere all'interno di un programma a lungo. Almeno finchè non viene inserito un input che "sollecita" l'istruzione che lo contiene

DEBUGGING

- Software testing **rivela i failure**
 - ovvero un comportamento anomalo del SUT
- **Debugging**: il processo usato per trovare un bug/fault e rimuoverlo
- Due fasi:
 - **fault localization/location**
 - es. individuare il file che contiene il fault
individuare la linea che contiene il fault
 - **fault removal**



PERCHÈ IL TESTING È DIFFICILE (1)?

- Testing **esaustivo** è “**non realizzabile**” nei casi reali
 - Es:
 - Testare completamente un software che esegue la somma di 2 interi a 32 bit
 - Occorre testare il programma **per tutti gli input possibili**, ovvero tutte le possibili combinazioni
 - 2^{64} input = *18.446.744.073.709.551.616*
 - 1 run al millisec = **584.942.417 anni**

2^{64}



00000000000000000000000000000000 + 00000000000000000000000000000000 = 00000000000000000000000000000000
000000000000000000000000000000001 + 00000000000000000000000000000000 = 000000000000000000000000000000001
⋮
000000000000000000000000000000001 + 000000000000000000000000000000001 = 000000000000000000000000000000010

1111111111111111111111111111111110 + 000000000000000000000000000000001 = 1111111111111111111111111111111111

PERCHÈ IL TESTING È DIFFICILE (2)?

- Il Testing è difficile perchè occorre selezionare **“pochi e buoni” input** nel dominio di tutti gli input possibili
 - “buoni” = con alta probabilità di trovare un fault ...



COME TROVARE BUONI INPUT?

- Due (macro) categorie:
 - Affidarsi ad approcci white box o black box



**BLACK
BOX**

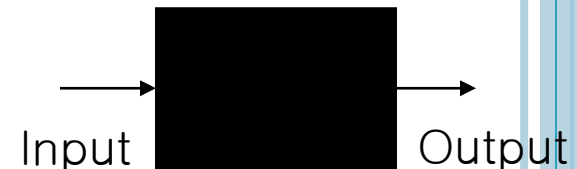
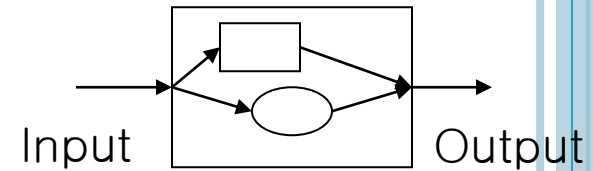
VS



**WHITE
BOX**

DUE “MACRO” CATEGORIE

- Il **white box testing** è basato su una conoscenza esplicita del SUT e della sua struttura
 - chiamato anche **structural testing**
 - *statement coverage testing* è un esempio
 - Lo scopo è quello di produrre abbastanza input per assicurare che ogni statement è eseguito almeno una volta
- Il **black box testing** non è basato su una conoscenza del SUT e della sua struttura
 - chiamato anche **functional testing**
 - di solito gli input sono generati a partire dai **requisiti/specifiche**

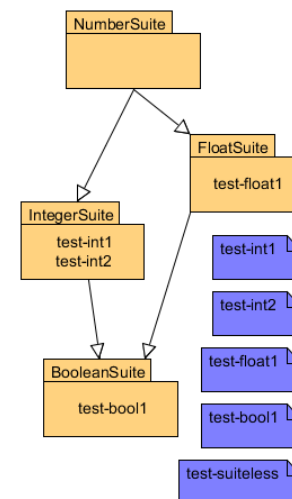


DEFINIZIONI DI TESTCASE E TESTSUITE

- Esistono diverse definizioni di testcase (**caso di test**)

IEEE Standard **Glossary** of Software Engineering Terminology

- A set of **inputs**, execution preconditions, and **expected outcomes** developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement [IEEE, Std 610.121990]
- Testsuite** = collezione di Testcases



ESEMPI (LA DEFINIZIONE VA ISTANZIATA)

Che cosa è “concretamente” un testcase dipende dal contesto:

Test case per desktop app “Sort”:

- Input: <“C”, 12, -29, 32, > C=Ordine crescente
- Output atteso: -29 12 32

input
+
output

Test case per Web Apps “Carrello”:

Input:

1. login alla pagina <http://www.abc.it/home.html>
2. vai sulla pagina “acquisti”
3. seleziona prodotto X
4. aggiungi il prodotto al carrello

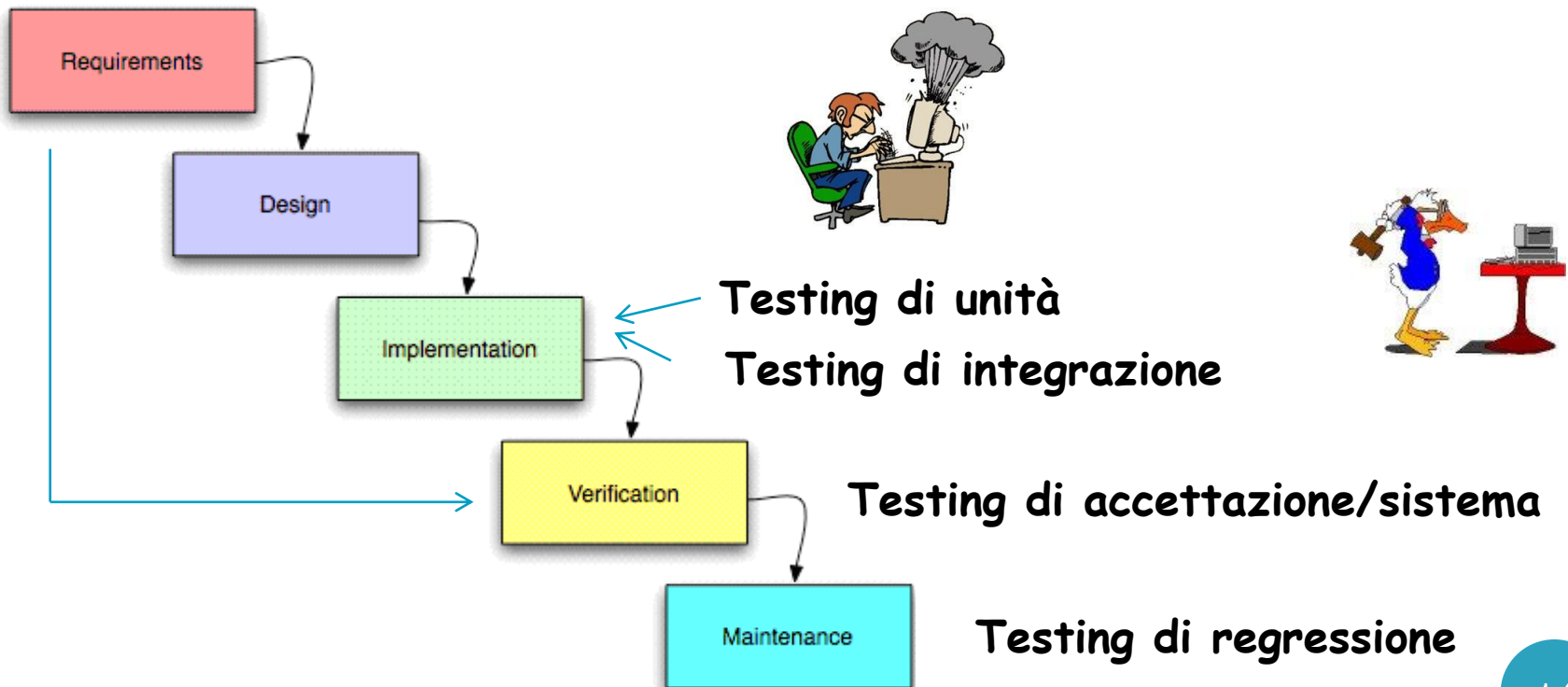
Output atteso:

Il prodotto X è stato aggiunto al carrello

sequence of steps
+
input
+
output

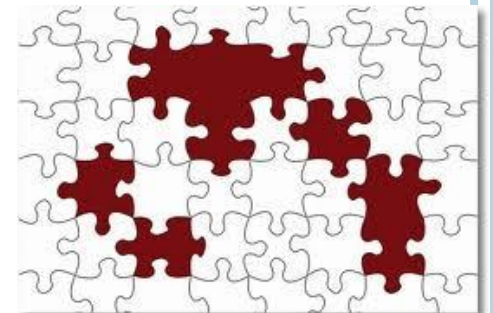
TIPOLOGIE DI TESTING (O LIVELLI)

- In relazione alla fase di sviluppo in cui viene condotto il testing:



TIPOLOGIE DI TESTING (1)

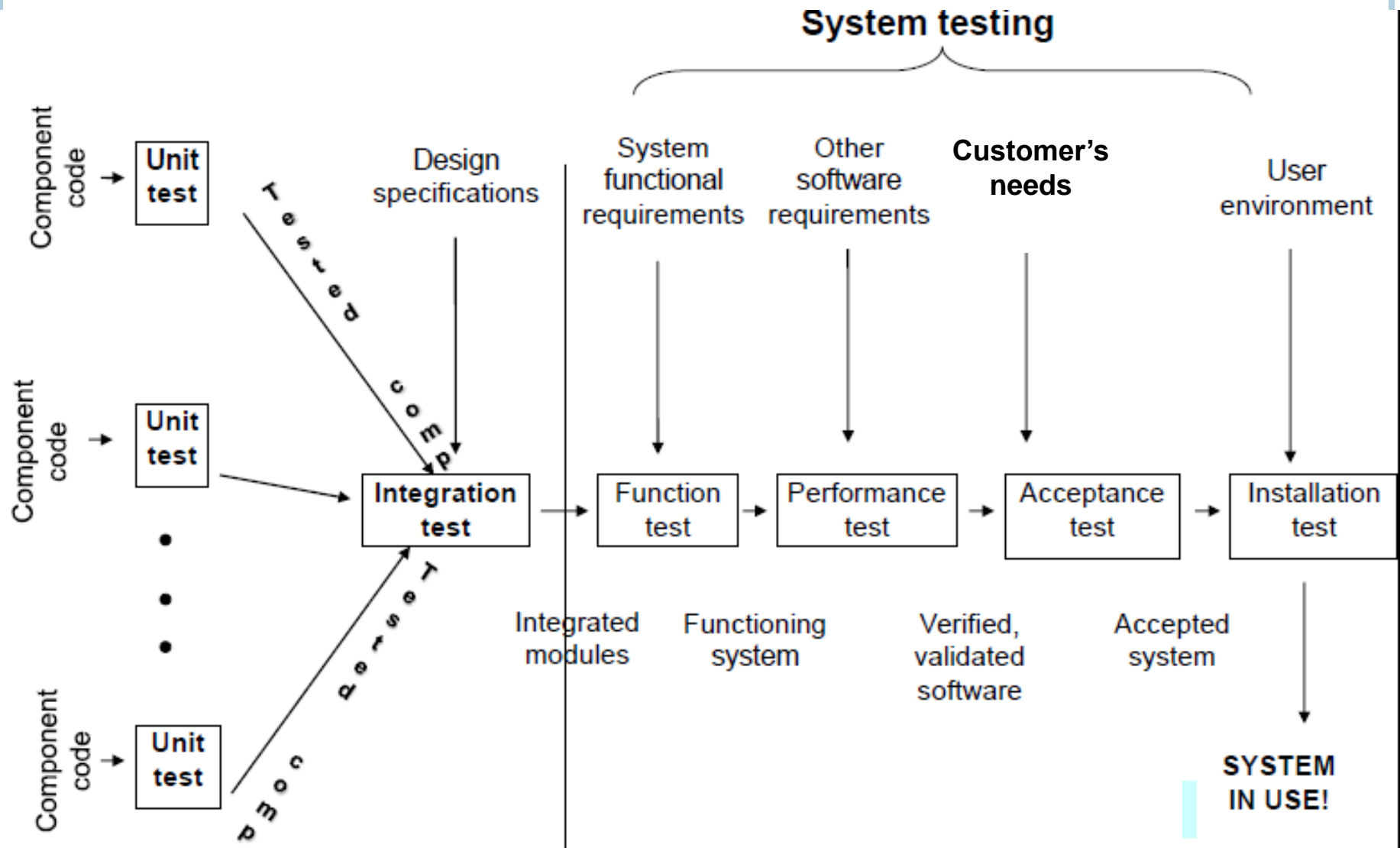
- (Fase Implementazione) **Testing di unità**
 - L'unità (funzione, procedura, classe, ...) viene testata dallo **sviluppatore** *isolandola il più possibile dal resto*
 - Concetto di stub / mock object (vedi TAP)
 - Di solito approccio **white-box**
- (Fase Implementazione/Integrazione) **Testing di integrazione**
 - I moduli/componenti sono “testati/e assieme”
 - Viene testato come comunicano i moduli/componenti e quali info vengono scambiate
 - **Interfacce** tra moduli di solito sono **fonti di errori**
 - Ordine sbagliato dei parametri
 - Precondizioni non rispettate (Design by contract)
 - Protocollo di comunicazione non rispettato
 -



TIPOLOGIE DI TESTING (2)

- (Fase di Testing di sistema) **Testing di sistema**
 - Il “**Testing group**” verifica che il software soddisfa i requisiti e anche i bisogni dell’utente (**acceptance testing**)
 - Di solito **black-box**
 - Non solo aspetti di correttezza ma anche **performance**, **usabilità**, **security**, ...
- (Fase di Manutenzione) **Testing di regressione**
 - Tutte le volte che si effettua una modifica in un applicazione c’è il rischio di **side effect** in zone del codice che apparentemente non sembrerebbero impattate dalla modifica
 - Scopo del test di regressione è verificare che **non ci siano state delle regressioni (o side effects)**

METTENDO TUTTO ASSIEME ...



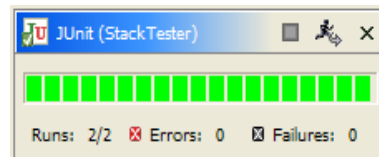
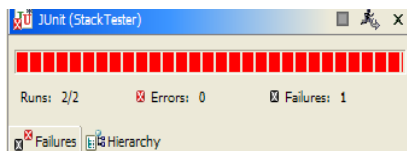
TESTING MANUALE VS. TEST AUTOMATIZZATO

- Nel testing automatizzato I tester implementano dei test script (**porzioni di codice**) che sono eseguiti da un framework che dopo la loro esecuzione riporta il risultato (pass/fail)
 - Vedere TAP, SSGS e FSTT



COSA è JUnit?

- E' un **framework di testing** per programmi **Java**
 - *Unit testing*
 - Framework Java = insieme di classi e convenzioni per usarle
 - Largo utilizzo di annotazioni (@Test)
- Sviluppato da:
 - *Erich Gamma*
 - Anche Design Pattern e Eclipse ...
 - *Kent Beck*
 - Anche Extreme Programming e TDD ...
- Integrato in **Eclipse** attraverso un plug-in grafico



Esempio JUnit

Framework Junit Jupiter



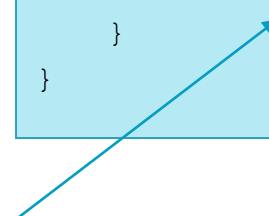
```
1 package code;
2
3 public class Calculator {
4
5     public Calculator() {
6         // TODO Auto-generated constructor stub
7     }
8
9     public int sum(int a, int b) {
10         return a+b;
11     }
12
13     public int sub(int a, int b) {
14         return a-b;
15     }
16 }
17
```

```
import static
org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

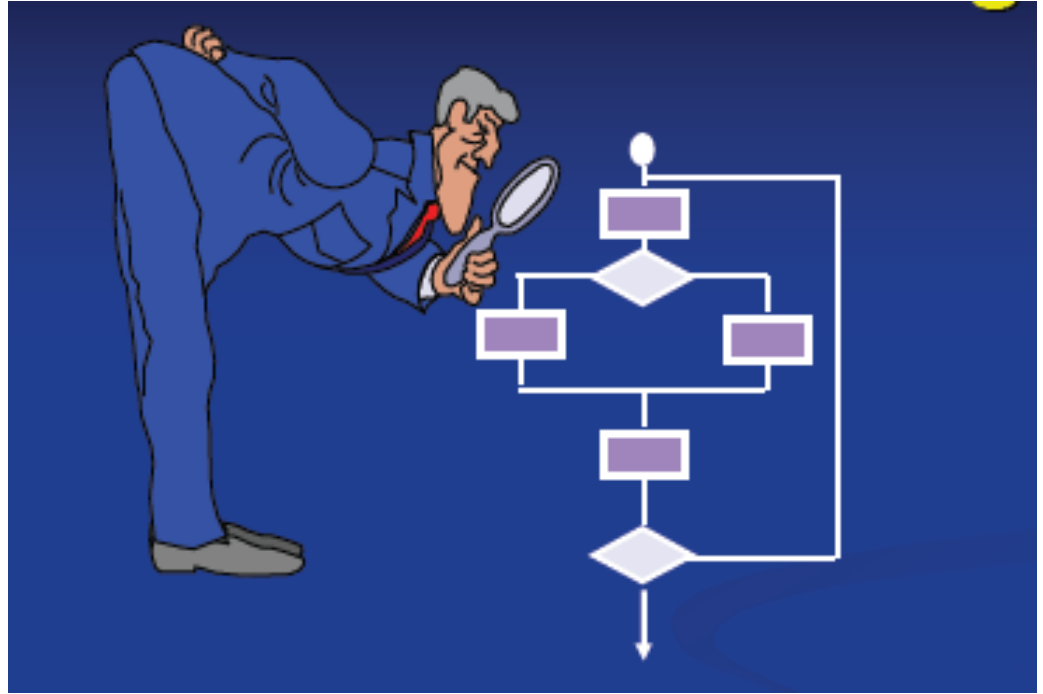
public class CalculatorTest {
    Calculator c;

    @Test
    public void testSum() {
        c = new Calculator();
        System.out.println(''testSum'');
        assertEquals(3, c.sum(2, 1));
    }

    @Test
    public void testSub() {
        c = new Calculator();
        System.out.println(''testSub'');
        assertEquals(3, c.sub(6, 3));
    }
}
```



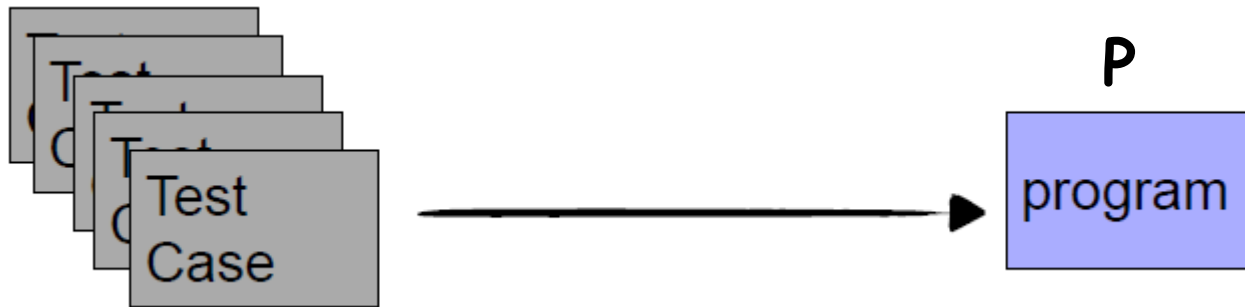
Asserzione



WHITE-BOX TESTING

CODE COVERAGE (COPERTURA)

Test suite T



- Di solito la qualità di una testsuite T si valuta misurando la **copertura** di T rispetto a P
 - Ad esempio: numero di linee di codice eseguite durante la fase di testing sul totale di linee di codice

Covered by tests →

Not covered by tests →

```
523 TEST(TaskSchedulerWorkerPoolTest, TestCodeCoverage) {
524     bool flag = true;
525     if (!flag) {
526         int value = 10;
527         EXPECT_EQ(10, value);
528     }
529     EXPECT_TRUE(flag);
530 }
```

ESEMPIO

- Unità di copertura: **linea di codice**

- È possibile contare il numero totale di linee di codice di P
- È possibile contare quelle realmente eseguite per ogni esecuzione

- **TestSuite 1** = {(x=5)}
 - Coverage = $7/8 = 87,5\%$

- **TestSuite 2** = {(x=5), (x=22)}
 - Coverage = $8/8 = 100\%$

```
1 scanf("%d", &x);
2 a = x + 1;
3 b = x - 1;
4 if (a < 10)
5     x++;
6 if (b > 20)
7     x--;
8 printf("%d\n", x);
```

```
1 scanf("%d", &x);
2 a = x + 1;
3 b = x - 1;
4 if (a < 10)
5     x++;
6 if (b > 20)
7     x--;
8 printf("%d\n", x);
```

La Testsuite 2 è "più buona" della Testsuite 1!

 Executed

DIVERSI MODI DI MISURARE LA COPERTURA

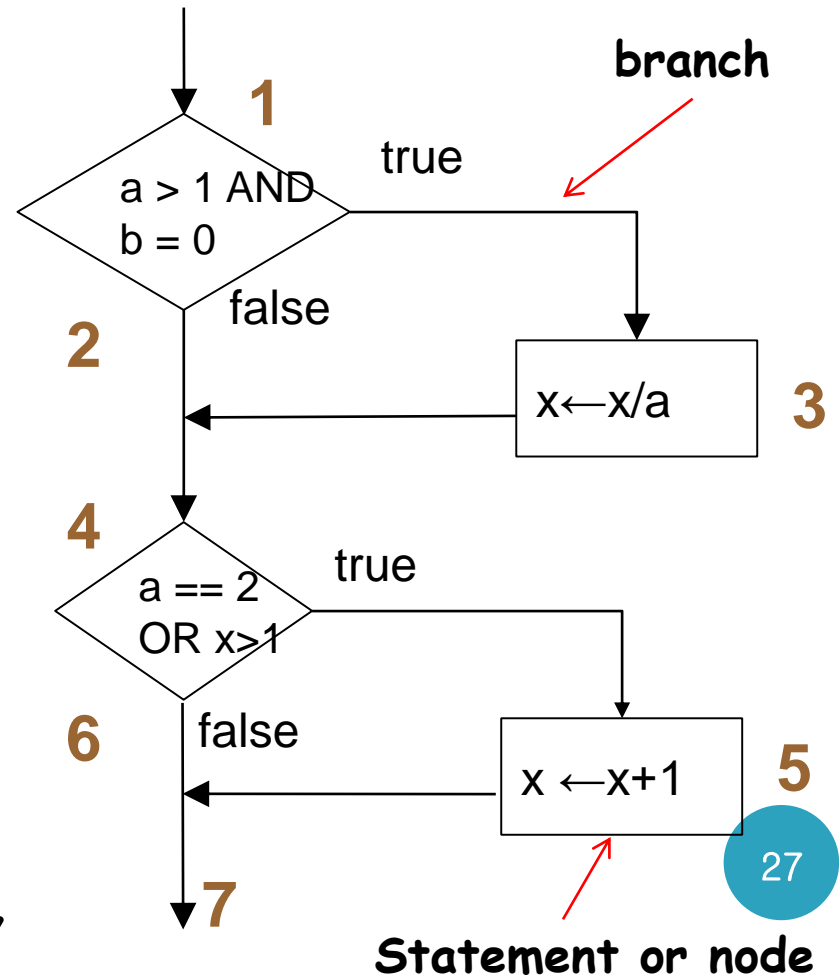
- La **Coverage** può essere basata su:
 - Codice sorgente (o codice intermedio)
 - LOC
 - **Modelli** (generati a partire dal codice)
 - Grafo di flusso di controllo (control flow graph)
 - State machines
 - E.g., per codice OO
 - Data flow graph
 - ...
 - Requisiti e Specifiche (black box testing)

CONTROL FLOW GRAPH (CFG)

Rappresenta mediante **un grafo** tutti i possibili **cammini** che possono essere attraversati durante l'esecuzione di P

```
int proc(int a, int b, int x)
{
    if ((a>1) && (b==0)) // 1
    {
        x = x/a; // 3
    }
    if ((a==2) || (x>1)) // 4
    {
        x = x+1; // 5
    }
    return x; // 7
}
```

Path (cammino)

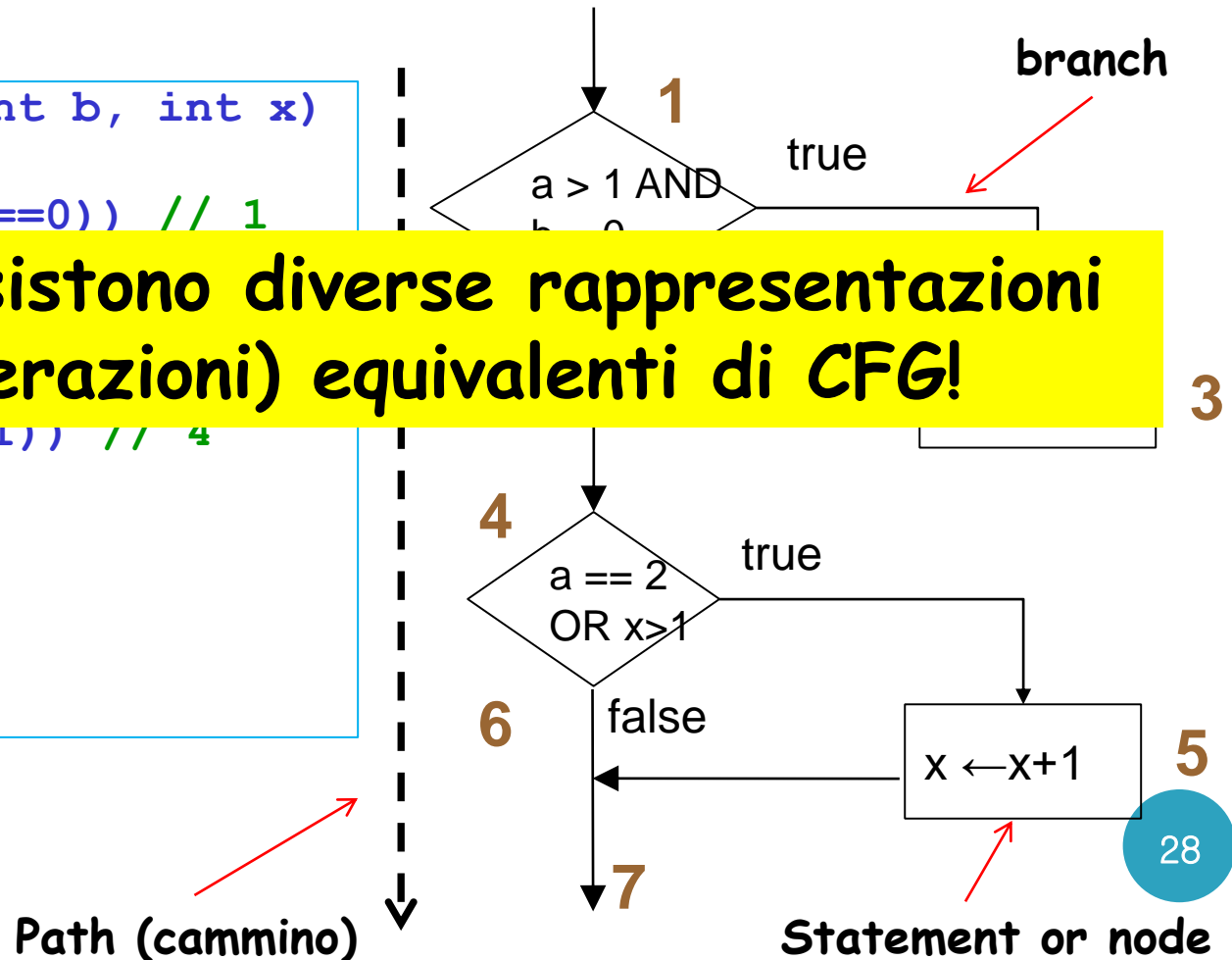


CONTROL FLOW GRAPH (CFG)

Rappresenta mediante **un grafo** tutti i possibili **cammini** che possono essere attraversati durante l'esecuzione di P

```
int proc(int a, int b, int x)
{
    if ((a>1) && (b==0)) // 1
    {
        if ((a==2) || (x>1)) // 4
        {
            x = x+1; // 5
        }
        return x; // 7
    }
}
```

Warning: Esistono diverse rappresentazioni (e numerazioni) equivalenti di CFG!



CRITERI DI COPERTURA BASATI SU CFG

- **Statement (node) coverage**
- **Branch coverage**
 - Anche chiamato decision coverage
 - Copertura minima richiesta da “IEEE unit test standard”
 - ANSI/IEEE Std 1008-1987
- **Multiple Condition Coverage (MCC)**
 - Copre tutte le combinazioni possibili delle condizioni nei nodi decisionali
 - If (x=6) && (y=7) then
- **All Paths coverage**
 - Copertura di tutti i cammini del CFG
 - 100% path coverage è **impossibile in pratica**
 - Ci sono i Loop ...



Power

IN PRATICA

○ Si sceglie un criterio

- Seguendo gli “standard” o le prescrizioni aziendali
 - di solito **branch coverage** ...
- Considerando che, in generale:
 - + è la power + è complesso trovare i casi di test
 - + è la power + aumenta la dimensione della testsuite
 - + è la power + aumentano i costi e i tempi di esecuzione
 - + è la power “+ è probabile” che la testsuite riveli dei fault

Compromesso!!

○ Si sceglie una copertura

- Es. 75%



○ Si crea una testsuite seguendo il criterio scelto con tanti casi di test al fine di raggiungere la copertura voluta ...

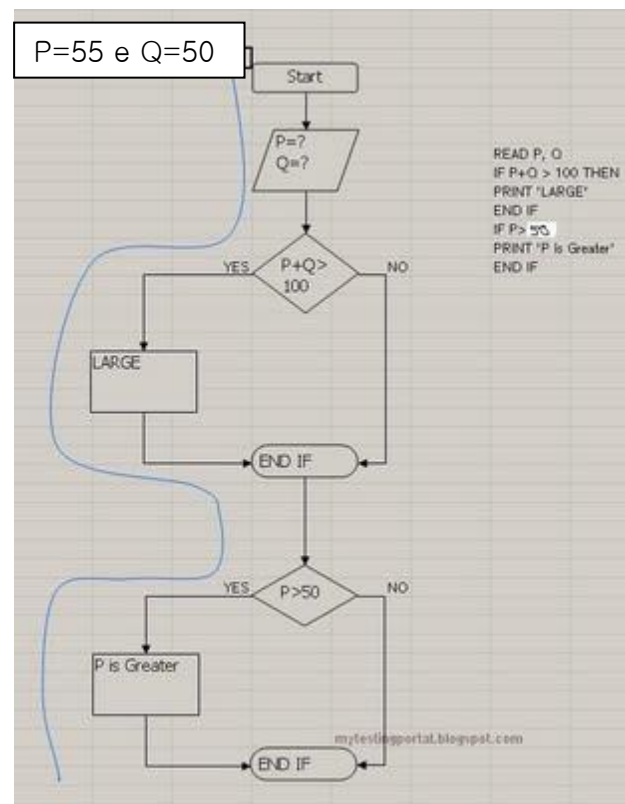
STATEMENT COVERAGE

Criterio: Tutti gli **statement** (o nodi) devono essere coperti (sollecitati) durante l'esecuzione della Testsuite

- E' il criterio **+ debole** di copertura
 - Alcuni branch non sono coperti!

Procedura:

- Selezionare alcuni path che coprono tutti gli statement
- Scegliere gli input in modo da percorrere i path selezionati



STATEMENT COVERAGE (ESEMPIO)

```
int proc(int a, int b, int x)
{
    if ((a>1) && (b==0)) // 1
    {
        x = x/a; // 3
    }
    if ((a==2) || (x>1)) // 4
    {
        x = x+1; // 5
    }
    return x; // 7
}
```

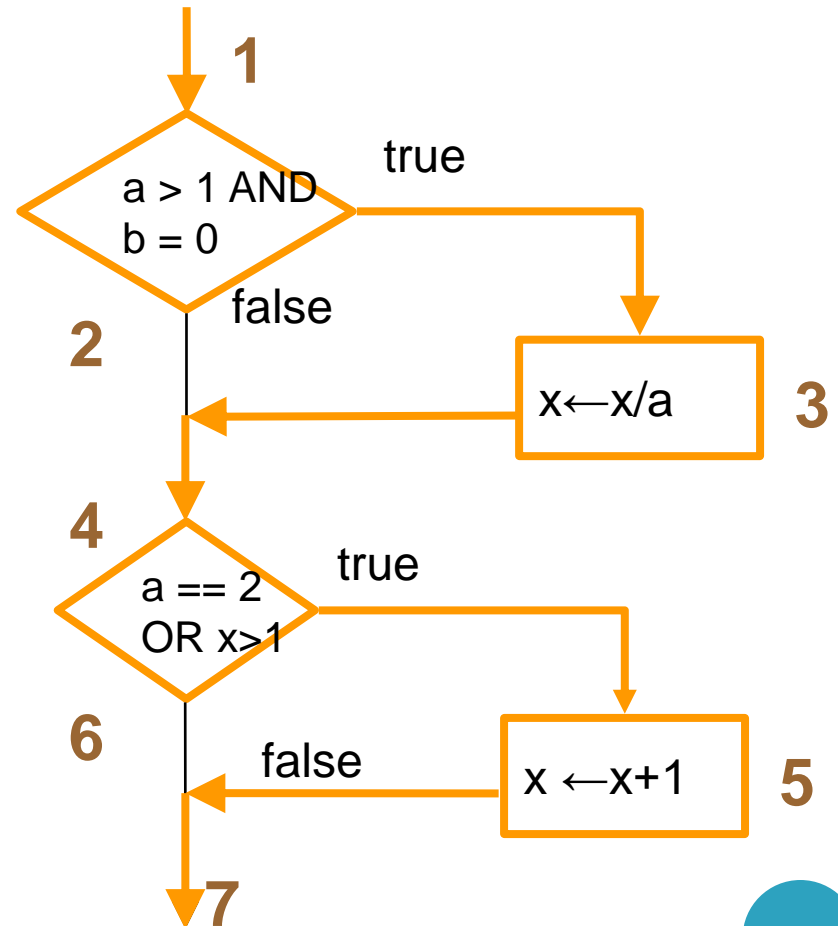
- Il seguente path è sufficiente per statement coverage:

1 - 3 - 4 - 5 - 7

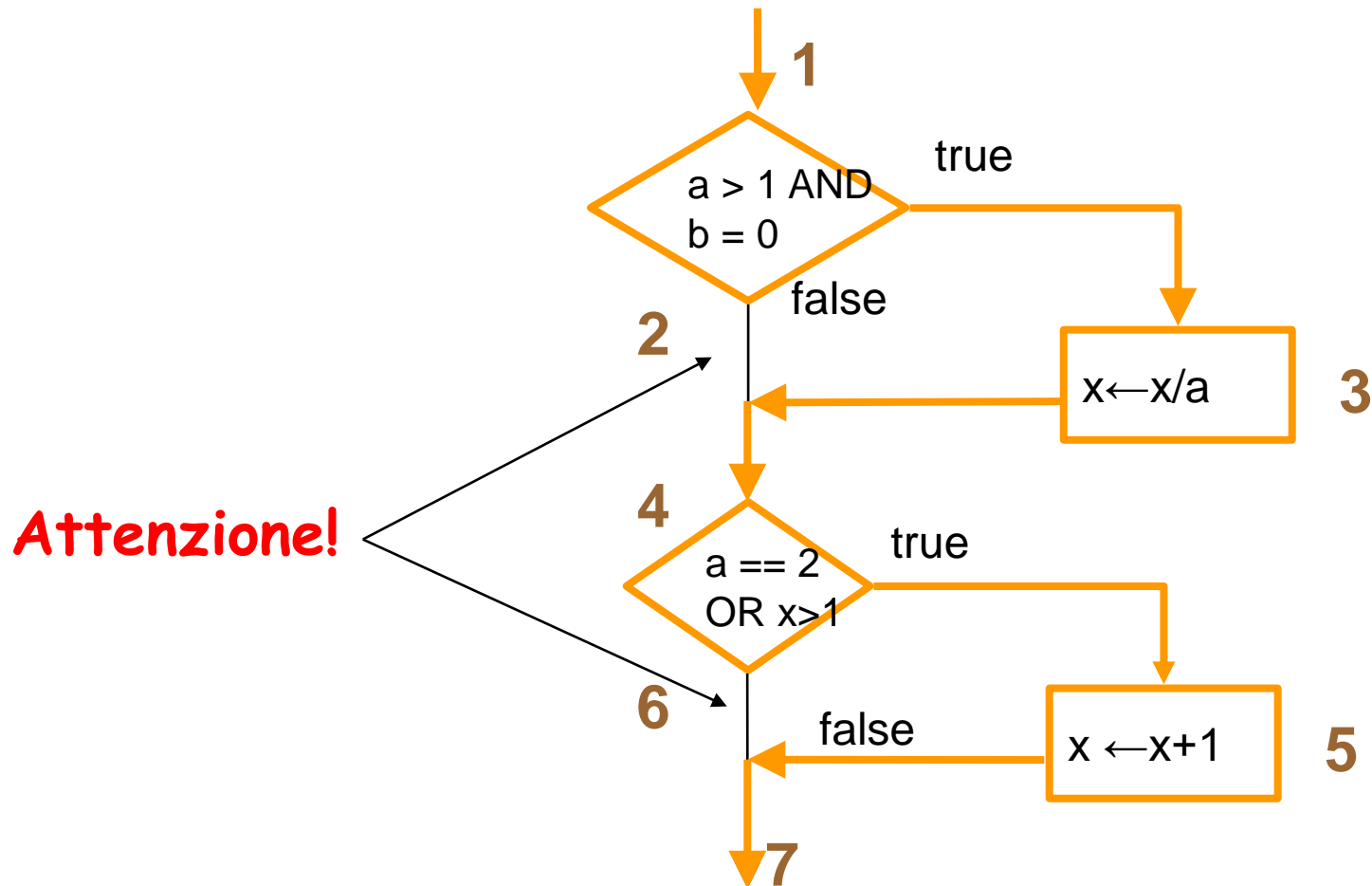
```
int proc(int a, int b, int x)
```

Input possibile:

a = 2, b = 0, x = 4



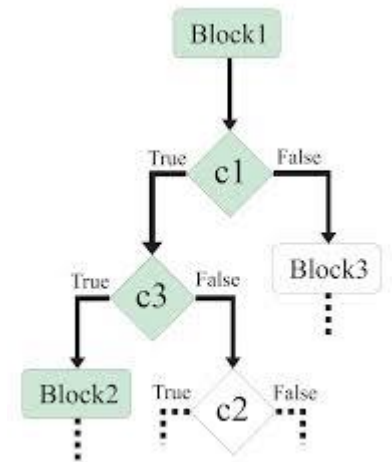
BRANCH NON COPERTI?



BRANCH COVERAGE

Criterio: Tutti i **branch** devono essere coperti (sollecitati) durante l'esecuzione della testsuite

- Cioè:
 - I branch “true” e “false” degli **if** statement
 - Ogni “case” in uno statement **switch**
 - Loop
 - **While, For, Goto, ...**



Procedura:

- Selezionare alcuni path che coprono tutti i branch
- Scegliere gli input in modo da percorrere i path selezionati

Branch coverage \Rightarrow **Statement coverage!**

BRANCH COVERAGE (ESEMPIO)

```
int proc(int a, int b, int x)
{
    if ((a>1) && (b==0)) // 1
    {
        x = x/a; // 3
    }
    if ((a==2) || (x>1)) // 4
    {
        x = x+1; // 5
    }
    return x; // 7
}
```

- I seguenti path sono sufficienti per branch coverage:

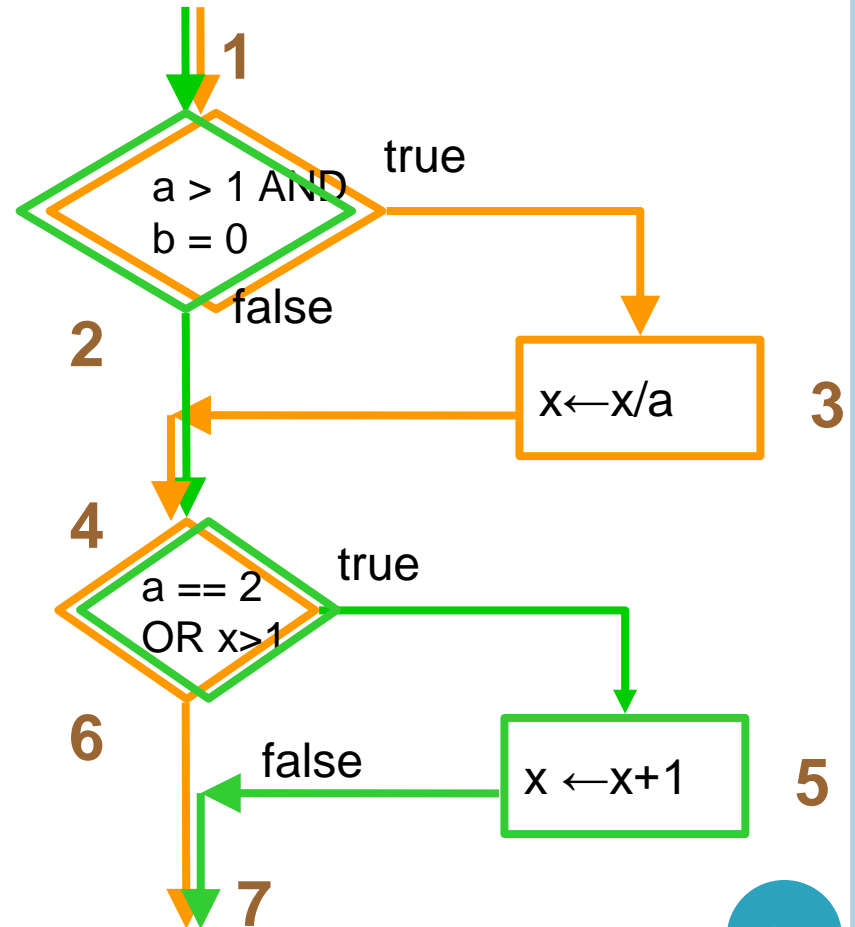
1 - 2 - 4 - 5 - 7 ———

1 - 3 - 4 - 6 - 7 ———

Input possibili:

1. a = 2, b = 2, x = -1 ———

2. a = 3, b = 0, x = 1 ———



MULTIPLE CONDITION COVERAGE (MCC)

MCC coverage \Rightarrow **Statement e Branch coverage!**

○ Criterio:

- Ogni **condizione atomica** (cioè che non contiene AND e OR) deve essere coperta (occorrono degli input che la rendono true e false)
 - Es. If A then print C
 - TEST CASE1 A=true
 - TEST CASE2 A=false
- In una **condizione composta** (cioè che contiene AND e/o OR), **ogni combinazione** delle condizioni atomiche deve essere coperta durante l'esecuzione dei casi di test
 - Es.

```
if (A||B)
then
print C
```

```
TEST CASE1: A=TRUE, B=TRUE
TEST CASE2: A=TRUE, B=FALSE
TEST CASE3: A=FALSE, B=TRUE
TEST CASE4: A=FALSE, B=FALSE
```

MCC (ESEMPIO) - 1

```
int proc(int a, int b, int x)
{
    if ( (a>1) && (b==0) )
    {
        x = x/a;
    }
    if ( (a==2) || (x>1) )
    {
        x = x+1;
    }
    return x;
}
```

Dobbiamo soddisfare i seguenti casi:

1. **a > 1** is **true** and **b = 0** is **true**
2. **a > 1** is **true** and **b = 0** is **false**
3. **a > 1** is **false** and **b = 0** is **true**
4. **a > 1** is **false** and **b = 0** is **false**
5. **a = 2** is **true** and **x > 1** is **true**
6. **a = 2** is **true** and **x > 1** is **false**
7. **a = 2** is **false** and **x > 1** is **true**
8. **a = 2** is **false** and **x > 1** is **false**

MCC (ESEMPIO) - 2

1. $a > 1$ is true and $b = 0$ is true
2. $a > 1$ is true and $b = 0$ is false
3. $a > 1$ is false and $b = 0$ is true
4. $a > 1$ is false and $b = 0$ is false
5. $a = 2$ is true and $x > 1$ is true
6. $a = 2$ is true and $x > 1$ is false
7. $a = 2$ is false and $x > 1$ is true
8. $a = 2$ is false and $x > 1$ is false

Input possibili:

$a = 2, b = 0, x = 2$ [1][5]

$a = 2, b = 1, x = 0$ [2][6]

$a = 0, b = 0, x = 2$ [3][7]

$a = 0, b = 1, x = 0$ [4][8]

ALL PATHS COVERAGE

Tutti i cammini del CFG devono essere coperti

- Insieme di tutti i path:

1 - 2 - 4 - 6 - 7

1 - 3 - 4 - 6 - 7

1 - 2 - 4 - 5 - 7

1 - 3 - 4 - 5 - 7

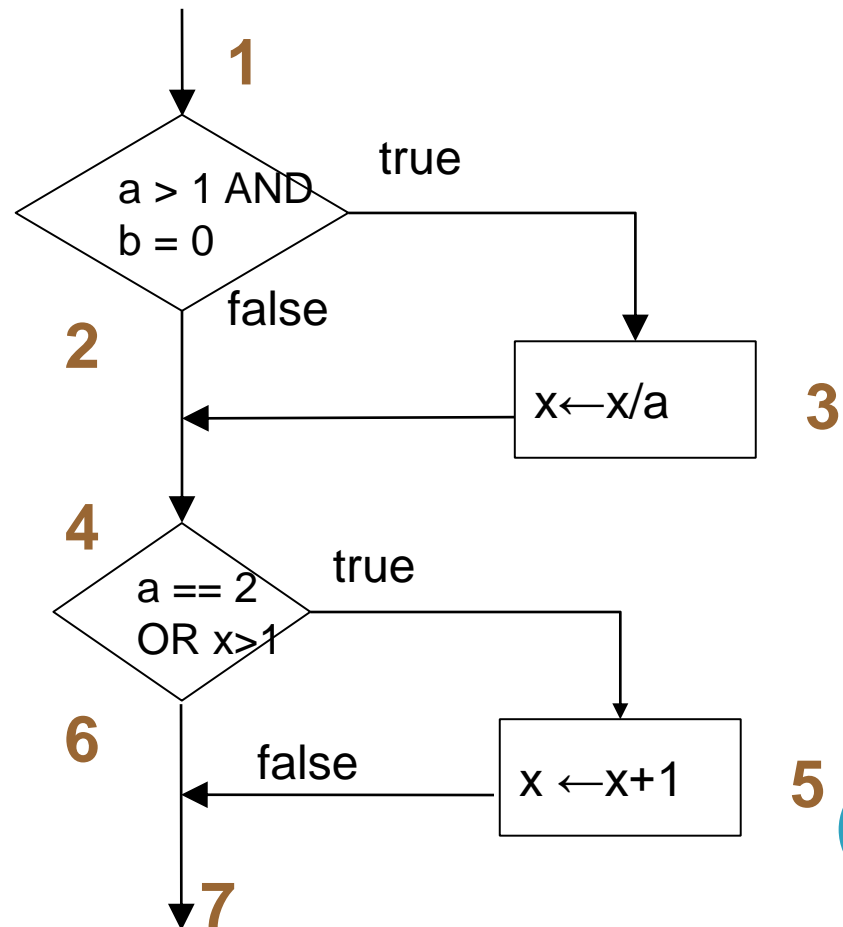
- Valori di input:

$a = 0, b = 1, x = 0$

$a = 3, b = 0, x = 0$

$a = 2, b = 1, x = 0$

$a = 2, b = 0, x = 0$



ALL PATHS COVERAGE: PROBLEMI

ATTENZIONE due problemi!

- “Non usabile” in pratica quando ci sono dei loop
 - **Troppi cammini!!!!**
 - Di solito si semplifica trattando i loop in modo binario:
 1. Il loop è eseguito (normalmente una volta)
 2. Il loop non viene eseguito
- Alcuni cammini potrebbero essere **“infeasible”**
 - Potrebbe essere che **non esiste** una combinazione di input che ci permetta di scegliere un particolare path
 - Vedi esempio dopo



ALL PATHS COVERAGE: PROBLEMI

ATTENZIONE due problemi!

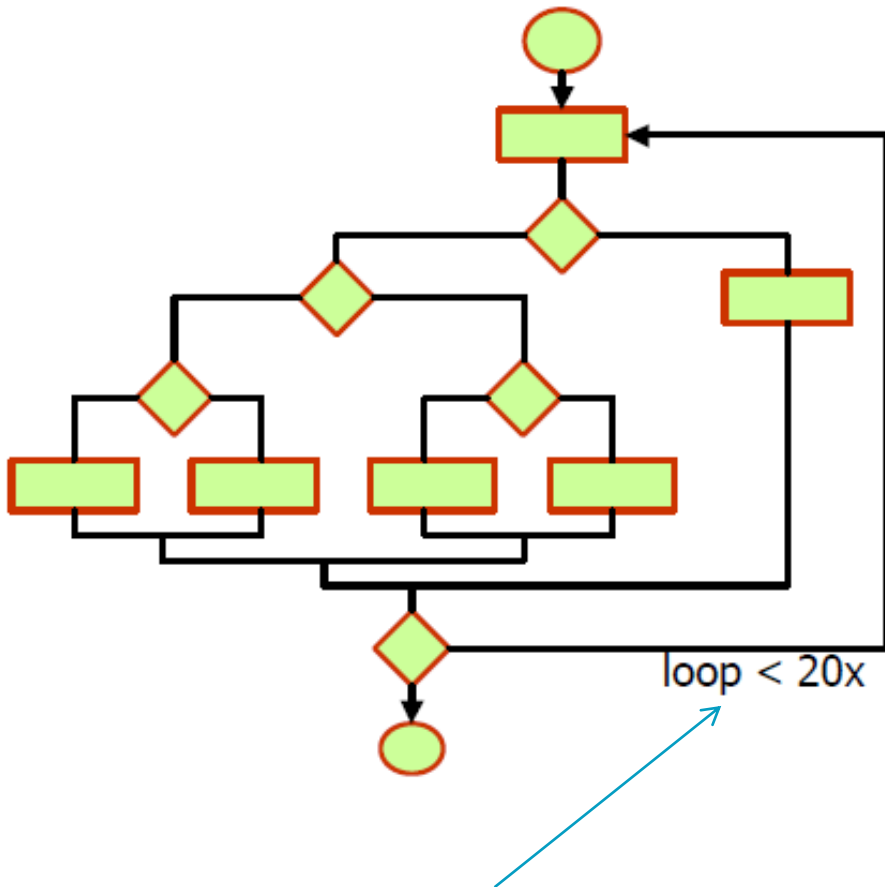
- “Non usabile” in pratica quando ci sono dei loop
 - **Troppi cammini!!!!**
 - Di solito si semplifica trattando i loop in modo binario:

Complicano l'applicazione del criterio

- 2. Il loop non viene eseguito
- Alcuni cammini potrebbero essere **“infeasible”**
 - Potrebbe essere che **non esiste** una combinazione di input che ci permetta di scegliere un particolare path
 - Vedi esempio dopo



NUMERO DI CAMMINI 'ASTRONOMICO'



- Numero di cammini possibili:

$$5^{20} + 5^{19} + 5^{18} + \dots + 5 \approx 10^{14}$$

≈ 100 trillion

- Se avessimo un computer in grado di eseguire un test al millisecondo, ci vorrebbero **3170 anni** solo per la loro esecuzione!
- Figuriamoci a progettarli ...

LOOP può essere eseguito fino a 20 volte a seconda dell'input

CAMMINI 'INFEASIBLE'

- Insieme di path per criterio "all paths":

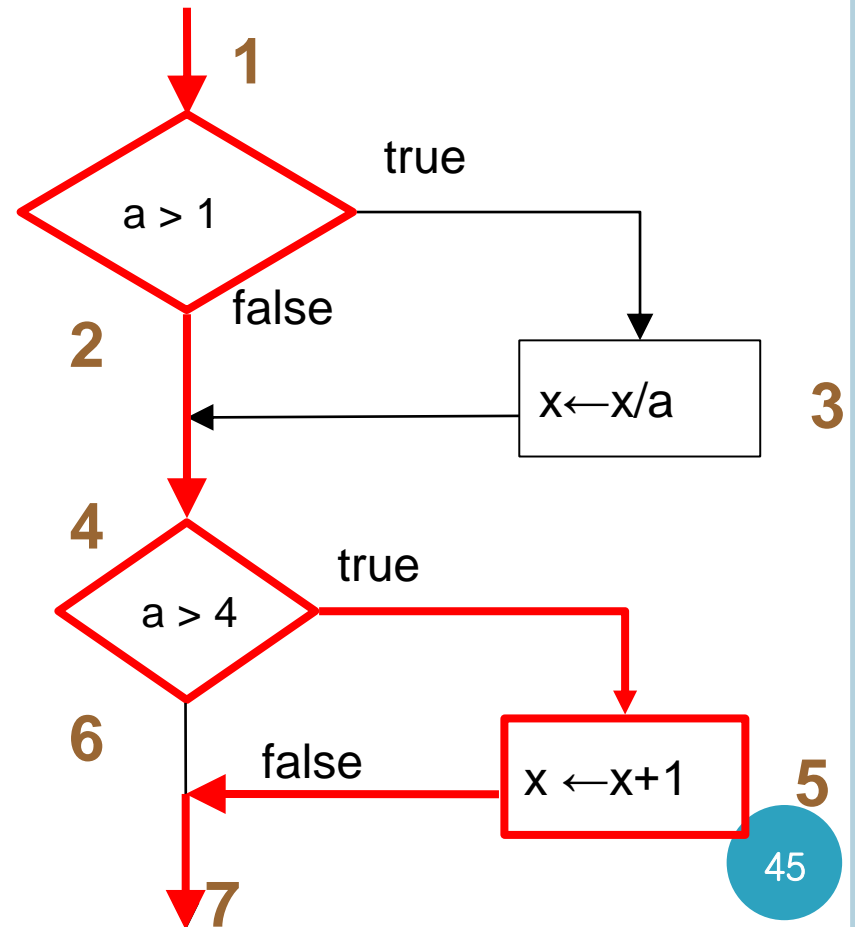
1 - 2 - 4 - 6 - 7

1 - 3 - 4 - 6 - 7

1 - 2 - 4 - 5 - 7

1 - 3 - 4 - 5 - 7

- Per poter coprire il cammino "rosso" dovremmo avere:
 - $a \leq 1$ AND $a > 4$ che è una contraddizione logica!



LIMITI DELLE TECNICHE WHITE-BOX (IN GENERALE)

- Spesso il numero di casi di test prodotti è molto grande (indipendentemente dal criterio scelto)
 - Si rivela essere **poco usabile in pratica!**
- Per questo motivo è spesso usato solo per **Unit testing** e per ‘piccole’ porzioni del sistema
 - “**Testing in the small**”
- Non è in grado di rivelare i fallimenti dovuti a “**missing feature errors**”
 - Possono essere scoperti solo considerando i requisiti/specifiche

TOOLS PER TESTING DI COPERTURA

Tool name	C++/C	Java	Other
Agitar [14]		X	
Bullseye [15]	X		
Clover [16]		X	.net
Cobertura [17]		X	
CodeTEST [18]	X		
Dynamic [19]	X		
EMMA [20]		X	
eXVantage [21]	X	X	
Gcov [22]	X		
Intel [23]	X		FORTTRAN
JCover [24]		X	
Koalog [25]		X	
Parasoft (C++test) [26]	X		
Parasoft (Jtest) [26]		X	
PurifyPlus [27]	X	X	Basic, .net
Semantic Designs (SD) [28]	X	X	C#, PHP, COBOL, PARLANSE
TCAT [29]	X	X	

JAVA:

- Clover:
 - <http://www.cenqua.com/clover>
- Emma:
 - <http://emma.sourceforge.net>
 - <http://www.eclemma.org>
- Coverlipse:
 - <http://coverlipse.sourceforge.net>
- Cobertura:
 - <http://cobertura.sourceforge.net>
-

Molti tool disponibili, segno che il concetto di copertura è importante

VISUAL STUDIO

Vedere TAP

TEST ARCHITETTURA ANALIZZA FIN...?

Esegui
Debug
Impostazioni test
Analizza code coverage
Finestre

Test selezionati
Tutti i test

Esplora test

Esegui tutto | Esegui...

Test superati(3)

- QuickNonZero 15 ms
- RootTestNeg... 13 ms
- SignatureTest 1 ms

Non analizzato

Analizzato

Attivare la colorazione

100 %

Risultati code coverage

ctsoasm_MAIN50531 2012-06-07 02...

Gerarchia	Non anal...	Non analizzato (%...	Anal...
ctsoasm_MAIN50531 201...	44	80.00%	11
fabrikam.math.dll	7	50.00%	7
{ } Fabrikam.Math	7	50.00%	7

Classe under test

```
public double SquareRoot(double x)
{
    if (x < 0.0)
    {
        throw new ArgumentOutOfRangeException();
    }
    double estimate = x;
    double previousEstimate = -x;
    while (System.Math.Abs(estimate - previousEstimate) >...)
    {

```

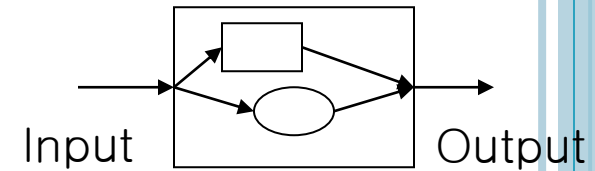


BLACK-BOX TESTING

DUE “MACRO” CATEGORIE

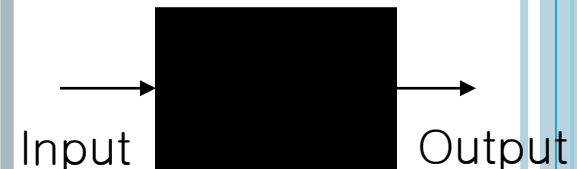
- Il **white box testing** è basato su una conoscenza esplicita del SUT e della sua struttura

- Chiamato anche **structural testing**
- Basato sul concetto di “copertura”
 - *Statement coverage testing* è un esempio



- Il **black box testing** non è basato su una conoscenza del SUT e della sua struttura

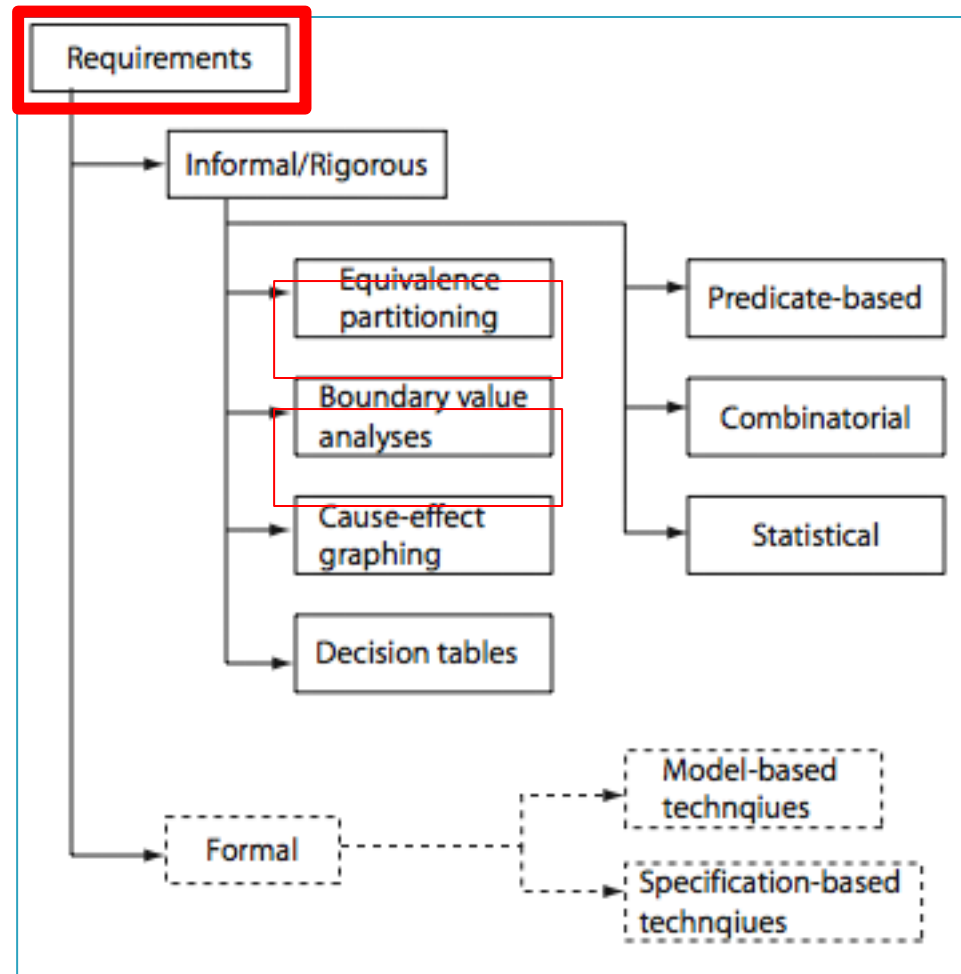
- Chiamato anche **functional testing**
- Di solito gli input sono generati a partire dai requisiti/specifiche



BLACK-BOX TESTING: VANTAGGI

- Il **black box testing** può essere usato con qualsiasi tipo di sistema **indipendentemente dalla tecnologia/piattaforma/linguaggio usato**
 - Applicazioni Web
 - Sistemi basati su microservizi
 - Applicazioni per Smartphone
 - ...
- Il **black box testing** può essere usato per **Unit, Integration e System testing**
 - In pratica usato soprattutto per System Testing!
- Gli sviluppatori possono scrivere casi di test **non appena i requisiti del sistema** sono disponibili
 - Cioè prima che il codice venga scritto ...

DIVERSE TECNICHE BLACK-BOX

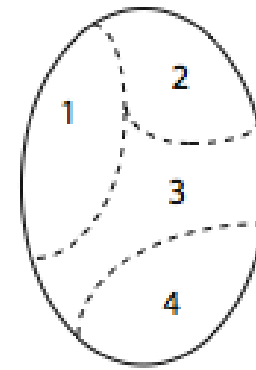


Tutte hanno come punto di partenza i requisiti (e/o documentazione del SW)

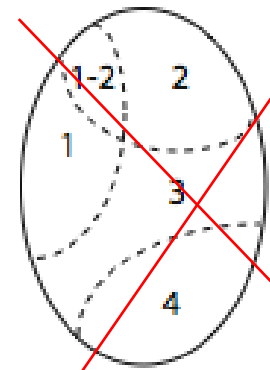
EQUIVALENCE PARTITIONING

- **Partizionamento dei dati di input**
- **Input domain** è suddiviso in classi t.c. il risultato risulta essere una **partizione**
- Le classi sono create assumendo che il SUT esibirà lo stesso **behaviour** su tutti gli elementi
 - Cioè sono “**classi di equivalenza**”
- A questo punto si sceglie un input per ogni classe e si testa il programma
 - **Input scelto è rappresentativo della classe!**

Input domain



(a)



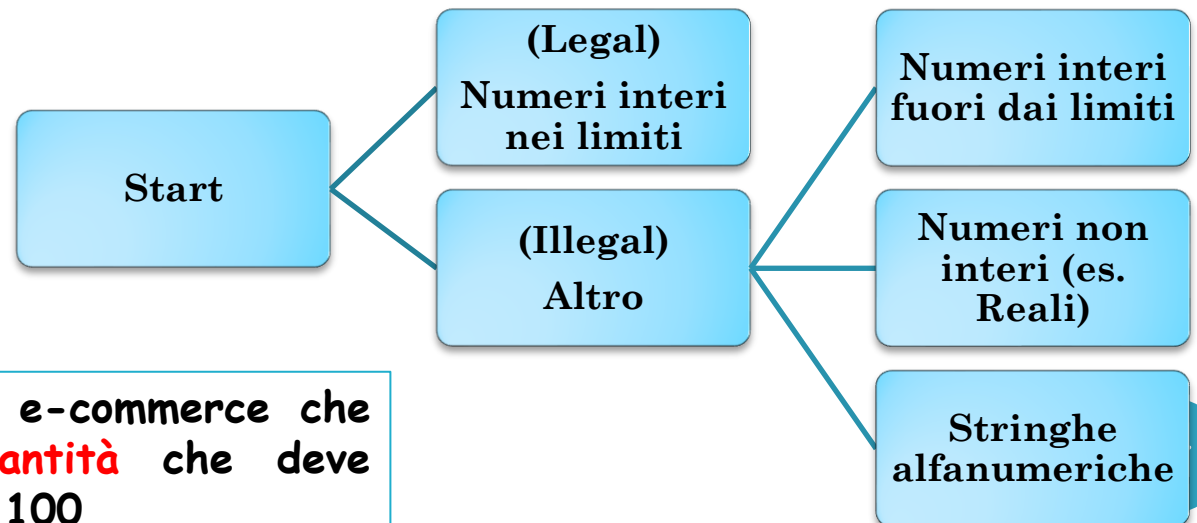
(b)

partizione

INPUT LEGALI E ILLEGALI ...

- Di solito il primo passo è quello di dividere l'**input domain** in due classi:
 - **Input legali** (expected, E)
 - **Input illegali** (unexpected, U)
 - Eccezioni o errori
- A loro volta queste classi possono ancora essere suddivise ...

Negative testing



Semplice applicazione di e-commerce che prende in input la **quantità** che deve essere compresa tra 1 -100

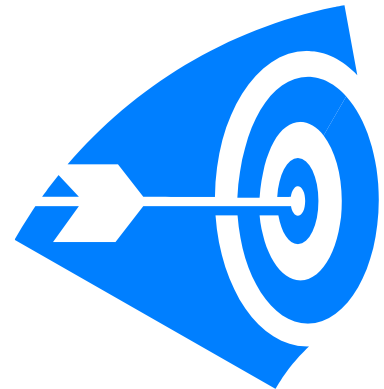
ESEMPIO: WORDCOUNT

- Consideriamo la funzione **wordCount** che:
 - Prende in input una stringa **w** e un filename **f**
 - Ritorna come output il numero di occorrenze di **w** nel testo del file **f**

stringa

file

Input: xy, [in xy vi sono xy tuple t.c.]
Output: 2



WORDCOUNT CLASSI DI INPUT

null - void - not void

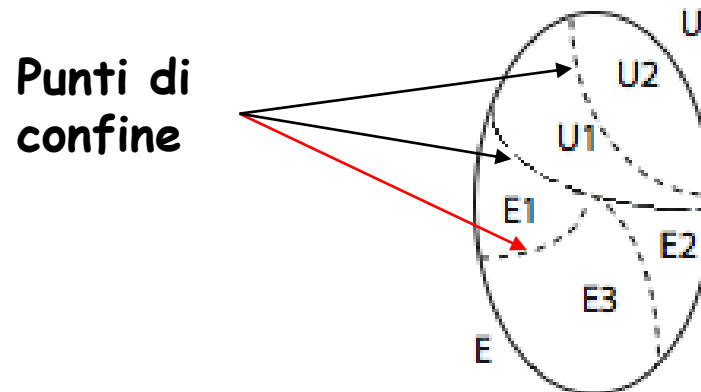
Does not exist - empty - not empty

Classi	w	f
E1	non-void	exists, not empty
E2	void	exists, not empty
E3	non-void	exists, empty
E4	void	exists, empty
U1	non-void	does not exist
U2	null	exists, empty
...

illegal

BOUNDARY VALUE ANALYSIS (BVA)

- **Boundary value analysis**: si scelgono i casi di test in prossimità della **frontiera (confini)** delle varie classi di equivalenza
 - Si applica dopo Equivalence Partitioning
- Si basa sul seguente **assunto**: è più probabile commettere errori vicino alla frontiera che non all'interno delle classi
 - Deriva dall'esperienza
 - Es. in una condizione è facile mettere “<n” al posto di un “<=n”
 - Provato da alcuni studi empirici ...



COME APPLICARE BVA?

1) Partizionare l'input domain (Equivalence Partitioning)

2) Identificare i confini per ogni partizione

3) Selezionare gli input in modo tale da comprendere i confini (e i punti vicini)

BVA: ESEMPIO CALCOLA PREZZO

- Semplice applicazione di **e-commerce**
- Vogliamo testare la funzionalità “**calcola prezzo**” che prende in input **codice** e **quantità** di un **prodotto**
- Supponiamo che il **codice** sia compreso nel range 99 ... 999 e la **quantità** nel range 1 ... 100

Codice = 127
Quantità = 3 ➡ Prezzo 87 euro



BVA: (1) CREAZIONE DELLE PARTIZIONI

- Semplice applicazione di **e-commerce**
- Vogliamo testare la funzionalità “**calcola prezzo**” che prende in input **codice** e **quantità** di un **prodotto**
- Supponiamo che il **codice** sia compreso nel range 99 ... 999 e la **quantità** nel range 1 ... 100

1. Creare le partizioni (classi di equivalenza)

Classi di equivalenza per codice:

E1: valori < 99

E2: valori nel range

E3: valori > 999

Classi di equivalenza per quantità:

E4: valori < 1

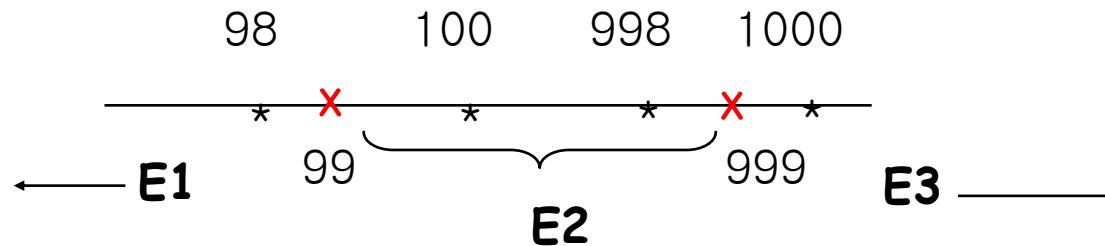
E5: valori nel range

E6: valori > 100

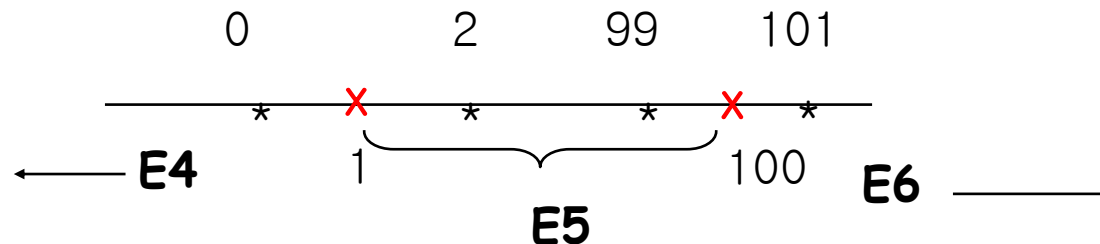


BVA: (2) IDENTIFICAZIONE DEI CONFINI

codice



quantità



- Partizioni (classi di equivalenza) **E1 ... E6** per **CalcolaPrezzo**
- I confini sono indicati con **X**
- I punti vicini ai confini sono indicati con *

BVA: (3) SELEZIONE INPUT (E CREAZIONE TESTSUITE T)

Includere i valori: **“su e vicino ai confini”**

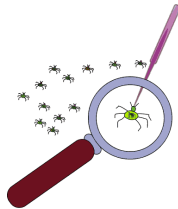
```
T={  t1: (code=98, qty=0),  
      t2: (code=99, qty=1),  
      t3: (code=100, qty=2),  
      t4: (code=998, qty=99),  
      t5: (code=999, qty=100),  
      t6: (code=1000, qty=101)  
}
```

I valori “illegali”
sono compresi

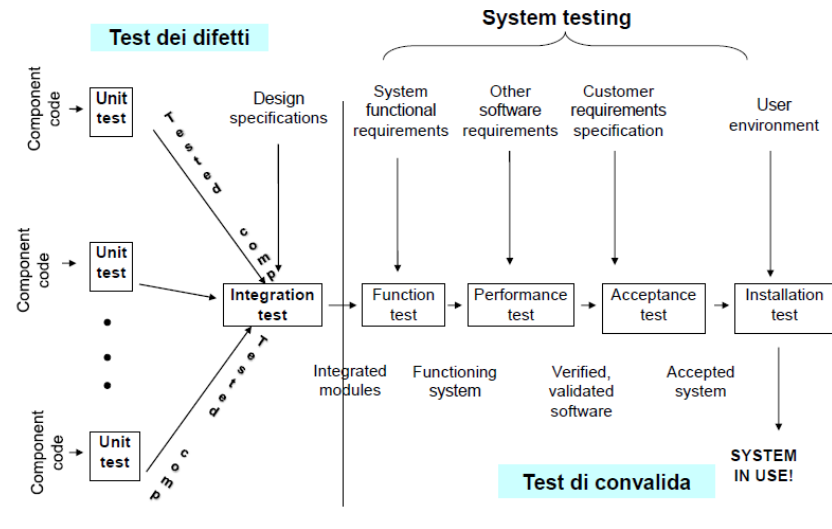
Codice 99 ... 999

Quantità 1 ... 100

RIASSUMENDO



Concetti: Software Testing e Debugging



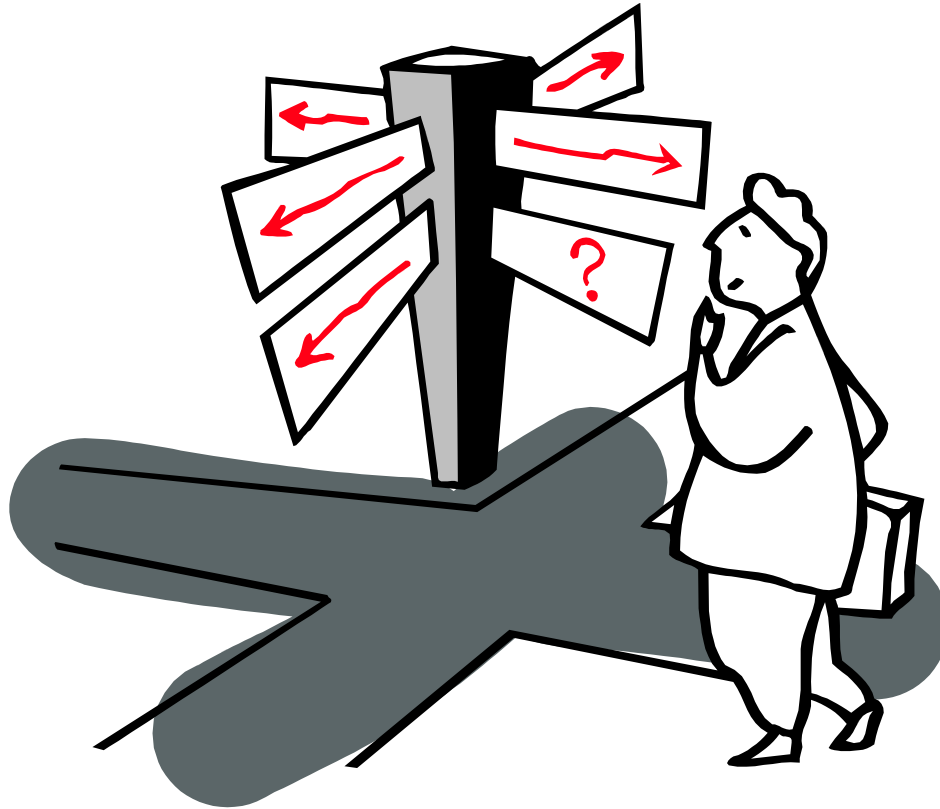
VS



BLACK
BOX

WHITE
BOX

THE END ...



Demande?