

Testing

TEPSIT

Verifica e Validazione

Verifica

Consiste nello stabilire/controllare se un software esegue le funzioni per le quali è stato realizzato in maniera corretta, senza malfunzionamenti.

Validazione

Consiste nel valutare se il software soddisfa i requisiti e le specifiche per esso stabilite;
Rientra nella validazione anche la valutazione del soddisfacimento dei requisiti di qualità.

Definizioni

- **Errore (Error)**
 - incomprensione umana nel tentativo di comprendere o risolvere un problema, o nell'uso di strumenti.
- **Difetto o anomalia (fault o bug)**
 - Manifestazione nel software di un errore umano, e causa del fallimento del sistema nell'eseguire la funzione richiesta.
- **Malfunzionamento o fallimento (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.

Testing, Debugging, Ispezione

Il **Testing** (collaudo) è un processo di esecuzione del software allo scopo di scoprirne i malfunzionamenti; Osservando i malfunzionamenti possiamo dedurre la presenza di difetti.

Il **Debugging** è il processo di scoperta dei difetti a partire dai malfunzionamenti rilevati.

L'**Ispezione** è un processo di analisi del software per scoprirne i difetti.

Adeguatezza dei test

Un test t rileva un malfunzionamento se il programma P non è corretto per t , ovvero se produce risultati diversi da quelli attesi.

- **t ha successo per un programma P se rileva uno o più malfunzionamenti presenti in P**

Un insieme di test T è *inadeguato* se esistono dei malfunzionamenti in P che il test T non è in grado di rilevare.

Tesi di Dijkstra

Il testing non può dimostrare l'assenza di difetti (**e dunque la correttezza del programma**), ma può solo dimostrare la presenza di difetti.

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

Un processo di testing deve consentire di acquistare fiducia nel software, mostrando che esso è pronto per l'uso operativo!

Test Ideale ed Esaustivo

Un test è **ideale** se l'insuccesso del test implica la correttezza del programma.

Un test **esaustivo** è un test che contiene tutti le combinazioni dei dati di ingresso al programma

- un test esaustivo è un test ideale;
- un test esaustivo non è pratico e quasi sempre non è fattibile.

Obiettivo realistico: selezionare casi di test che approssimano un test ideale.

Obiettivi del testing

- Deve aiutare a localizzare i difetti
Per facilitare il debugging.
- Dovrebbe essere ripetibile
Potrebbe non essere possibile se l'esecuzione del caso di test influenza l'ambiente di esecuzione senza la possibilità di ripristinarlo;
Potrebbe non essere possibile se nel software ci sono degli elementi indeterministici;
Ovvero dipendenti da input non controllabili.
- Dovrebbe essere preciso.

Valutazione dei risultati del test

Condizione necessaria per effettuare un test:

conoscere il comportamento atteso per poterlo confrontare con quello osservato.

L'oracolo conosce il comportamento atteso per ogni caso di prova.

- Oracolo umano
 - si basa sulle specifiche o sul giudizio.
- Oracolo automatico
 - generato dalle specifiche (formali);
 - stesso software ma sviluppato da altri;
 - versione precedente (test di regressione).

Terminazione del testing

Dal momento che il testing esaustivo è, in generale, irraggiungibile, altri criteri sono proposti per valutare quando il testing possa essere terminato.

- Criterio temporale: periodo di tempo predefinito;
- Criterio di costo: sforzo allocato predefinito;
- Criterio di copertura
 - Ad esempio provare almeno una volta tutte le righe del programma oppure tutti gli scenari descritti nei casi d'uso.
- Criterio statistico
 - Ad esempio terminare quando gli ultimi k test non hanno rilevato malfunzionamenti.

Selezione dei casi di test

Posto che non esiste (e non sapremmo riconoscere) l'insieme *ideale* di casi di test, la loro bontà si può misurare in termini di:

Efficacia

Numero di malfunzionamenti trovati / numero di malfunzionamenti da trovare.

Efficienza

Numero di test in grado di scoprire malfunzionamenti / numero di test totali.

Efficacia ed Efficienza

Per massimizzare entrambe, contemporaneamente, bisognerebbe trovare il massimo numero possibile di malfunzionamenti con il minimo numero possibile di casi di test.

In realtà l'efficacia in generale non è calcolabile perchè non conosciamo il numero totale di malfunzionamenti.

Molte strategie di testing riescono ad aumentare una delle due diminuendo l'altra.

L'efficacia va privilegiata quando si vuole un software affidabile.

L'efficienza va privilegiata se si vuole un testing meno costoso (in particolare se non può essere eseguito automaticamente).

Specifiche dei casi di test

Ogni caso di test, indipendentemente dalla sua tipologia, dovrebbe essere descritto quanto meno dai seguenti campi:

- Numero Identificativo
- Descrizione
 - Può indicare anche la funzionalità che si va testando
- Precondizioni
 - Asserzioni che devono essere verificate affinchè il test possa essere eseguito
- Valori di Input
- Valori di Output Attesi
 - Rappresentano l'oracolo
 - In alcune tipologie di test si fornisce una classe di equivalenza attesa per gli output anziché un singolo valore
- Postcondizioni Attese
 - Asserzioni assimiliabili agli output ma non verificabili direttamente dall'interfaccia utente.

All'atto dell'esecuzione del test, verranno aggiunti i seguenti campi:

- Output riscontrati
- Postcondizioni riscontrate
- Esito
 - Positivo (cioè malfunzionamento rilevato) se almeno un valore di output o una postcondizione riscontrati sono diversi da quelli attesi;
 - Negativo altrimenti.

Caratteristiche di un test case

Un test case dovrebbe sempre essere:

- Descritto in maniera univoca
- Ripetibile
- Dall'esito oggettivamente valutabile

Nella descrizione di un test case devono quindi esserci tutte le informazioni necessarie ad un tester (umano o automatico) per rieseguire identicamente il caso di test potendone valutare l'esito in maniera non ambigua.

Il processo di testing

- Test dei Componenti (o di Unità)
 - Testing di singole unità di codice (funzioni, oggetti, componenti riusabili);
 - È responsabilità di chi sviluppa il componente;
 - I test sono derivati in base all'esperienza dello sviluppatore.
- Test di Sistema
 - Testing di gruppi di componenti integrati per formare il sistema o un sotto-sistema;
 - È responsabilità di un team indipendente per il testing;
 - I test sono definiti in base alla specifica del sistema.

Testing di Unità

Il testing dei componenti (o test di unità) consiste nel testare i singoli componenti del sistema in isolamento.
È un processo di testing dei difetti.

Le unità da testare possono essere:

- Funzioni o metodi di un oggetto;
- Classi di oggetti con i loro attributi e metodi;
- Componenti composti che offrono una specifica interfaccia per accedere alle loro funzionalità.

Selezione dei casi di test

I casi di test (*test suite*) devono essere selezionati in modo da ottimizzare il più possibile efficacia ed efficienza.

Le due tipologie fondamentali di criteri di selezione sono:

- Testing Black Box;
- Testing strutturale (o White Box).

Testing Black Box

Si esercita il sistema immettendo input e osservando i valori degli output.

Non conosciamo (oppure non teniamo in conto):

- Il codice sorgente;
- Lo stato interno dell'applicazione;
- Il funzionamento interno dell'applicazione.

Viceversa, conosciamo e utilizziamo per la progettazione dei casi di test:

- L'interfaccia del sistema;
- La documentazione
 - In particolare i documenti relativi alla fase di analisi, ad esempio la descrizione degli scenari.

Criteri di copertura per il testing Black Box

Obiettivi principali del testing black box possono essere:

- La copertura degli scenari di esecuzione definiti a margine dei casi d'uso;
- La copertura delle funzionalità previste nella specifica dei requisiti.

Le difficoltà principali sono legate a:

- la ricerca di casi di test che coprano i dati scenari/funzionalità;
- la verifica dell'avvenuta copertura.

La soluzione più spesso adottata è quella di

- Generare casi di test con un opportuno criterio;
- Valutare l'effettiva copertura di scenari/funzionalità.

Suddivisione in Classi di equivalenza

Occorre dividere i possibili input in gruppi i cui elementi si ritiene che saranno trattati similarmente dal processo elaborativo.

Questi gruppi saranno chiamati classi di equivalenza.

Una possibile suddivisione è quella in cui classe di equivalenza rappresenta un insieme di stati validi o non validi per una condizione sulle variabili d'ingresso.

Chi esegue il testing eseguirà almeno un test per ogni classe di equivalenza.

Criterio di copertura delle classi di equivalenza.

Definizione delle classi di equivalenza

Se la condizione sulle variabili d'ingresso specifica:

- intervallo di valori
 - una classe valida per valori interni all'intervallo, una non valida per valori inferiori al minimo, e una non valida per valori superiori al massimo;
- valore specifico
 - una classe valida per il valore specificato, una non valida per valori inferiori, e una non valida per valori superiori;
- elemento di un insieme discreto
 - una classe valida per ogni elemento dell'insieme, una non valida per un elemento non appartenente;
- valore booleano
 - una classe valida per il valore TRUE, una classe non valida per il valore FALSE.

Inoltre, in ogni caso:

Una classe non valida per valore~~e~~ tipo

Selezione dei casi di test dalle classi di equivalenza

Ogni classe di equivalenza deve essere coperta da almeno un caso di test.

Un caso di test per ogni classe non valida.

Ciascun caso di test per le classi valide dovrebbe comprendere il maggior numero di classi valide ancora scoperte.

Esempio

Una condizione di validità per un input password è che la password sia una stringa alfanumerica di lunghezza compresa fra 6 e 10 caratteri.

Una classe valida CV1 è quella composta dalle stringhe di lunghezza fra 6 e 10 caratteri.

Due classi non valide sono:

- CNV2 che include le stringhe di lunghezza <6
- CNV3 che include le stringhe di lunghezza >10

Esercizio

Un programma C++ riceve in input una data, composta di giorno (numerico), mese (stringa che può valere gennaio ... dicembre), anno (numerico, compreso tra 1900 e 2000) e restituisce il giorno della settimana corrispondente (1= lunedì, 2= martedì ... 7=domenica).

Selezionare i casi di test mediante partizione in classi di equivalenza.

Le condizioni sull'input 'giorno'

Condizioni d'ingresso:

- Il giorno può essere compreso tra 1 e 31

Classi di equivalenza:

- **Valida**
 - CE₁ : $1 \leq \text{GIORNO} \leq 31$
- **Non valide**
 - CE₂ : $\text{GIORNO} < 1$
 - CE₃ : $\text{GIORNO} > 31$
 - CE₄ : GIORNO non è un numero intero

Le condizioni sull'input 'anno'

Condizioni di ingresso:

Deve essere compreso tra 1900 e 2000

Classi di equivalenza

Valida

CE₇: $1900 \leq \text{ANNO} \leq 2000$

Non valide

CE₈: $\text{ANNO} < 1900$

CE₉: $\text{ANNO} > 2000$

CE₁₀: ANNO non è un numero intero

Le condizioni sull'input 'mese'

Condizioni di ingresso:

Il mese deve essere nell'insieme M=(gennaio, febbraio, marzo, aprile, maggio, giugno, luglio, agosto, settembre, ottobre, novembre, dicembre).

Classi di equivalenza

Validi

CE₅: MESE ∈ M

Non valida

CE₆: MESE ∉ M

Scelta dei casi di test ...

Test case	TC1	TC2	TC3	TC4
Giorno	1	1	1	1
Mese	gennaio	gennaio	gennaio	gennaio
Anno	1980	1492	2018	duemila
Classi coperte	CE1, CE5, CE7	CE1, CE5, CE8	CE1, CE5, CE9	CE1, CE5, CE10
Test case	TC5	TC6	TC7	TC8
Giorno	1	0	35	primo
Mese	brumaio	gennaio	gennaio	gennaio
Anno	1980	1980	1980	1980
Classi coperte	CE1, CE6, CE7	CE2, CE5, CE7	CE3, CE5, CE7	CE4, CE5, CE7

Dimensione della Test Suite

Ogni TC riesce a coprire almeno una classe di equivalenza non coperta da alcuno dei precedenti.

La Test Suite comprendente TC1...TC8 copre tutte le classi di equivalenza (ma non tutte le possibili combinazioni ...)

Ad esempio, non viene testata la risposta del sistema ad una data di nascita come il 30 febbraio!

Per valutare la qualità della test suite bisognerebbe valutare contemporaneamente:

- L'efficienza, in termini di casi di test che riescono a scoprire nuovi malfunzionamenti;
- L'efficacia, in termini di malfunzionamenti trovati.

Nel nostro caso, proporre casi di test in grado di sollecitare tutte le combinazioni ammissibili degli input farebbe aumentare probabilmente l'efficacia della test suite riducendo l'efficienza.

Scrittura dei test di unità

Il testing a livello di unità dei comportamenti di una classe dovrebbe essere progettato ed eseguito dallo sviluppatore della classe, contemporaneamente allo sviluppo stesso della classe.

Di questa opinione sono in particolare Erich Gamma e Kent Beck, meglio conosciuti come gli autori dei Design Pattern e dell'eXtreme Programming (che verrà presentato nella lezione dedicata ai cicli di vita).

Vantaggi:

Lo sviluppatore conosce esattamente le responsabilità della classe che ha sviluppato e i risultati che da essa si attende.

Lo sviluppatore conosce esattamente come si accede alla classe, ad esempio:

- Quali precondizioni devono essere poste prima di poter eseguire un caso di test;
- Quali postcondizioni sui valori dello stato degli oggetti devono verificarsi.

Svantaggi:

Lo sviluppatore tende a difendere il suo lavoro ... troverà meno errori di quanto possa fare un tester!

Testing Automation

Se la progettazione dei casi di test é un lavoro duro e difficile, l'esecuzione dei casi di test é un lavoro noioso e gramo!

L'automatizzazione dell'esecuzione dei casi di test porta innumerevoli vantaggi:

- **Tempo risparmiato** (nell'esecuzione dei test);
- **Affidabilità** dei test (non c'è rischio di errore umano nell'esecuzione dei test)
 - Si può migliorare l'efficacia a scapito dell'efficienza sfruttando il fatto che l'esecuzione dei test è poco costosa;
- **Riuso (parziale)** dei test a seguito di modifiche nella classe.

Testing basato su main

Scrivere un metodo di prova ("main") in ogni classe contenente del codice in grado di testare i suoi comportamenti.

Problemi

- Tale codice verrà distribuito anche nel prodotto finale, appesantendolo.
- Come strutturare i test case? Come eseguirli separatamente?

Per tutti questi motivi, cerchiamo un approccio sistematico:

- Che separi il codice di test da quello della classe;
- Che supporti la strutturazione dei casi di test in test suite;
- Che fornisca un output separato dall'output dovuto all'esecuzione della classe.

Famiglia X-Unit

La soluzione alle problematiche precedenti è data dai framework della famiglia X-Unit:

- JUnit (Java)
 - È il capostipite; fu sviluppato originariamente da Erich Gamma and Kent Beck
- CppUnit (C++)
- csUnit (C#)
- NUnit (.NET framework)
- HttpUnit (Web Application)

JUnit è un framework (in pratica consiste di un archivio .jar contenente una collezione di classi) che permette la scrittura di test in maniera ripetibile.

Componenti di un test XUnit

Una classe di test, contiene:

- Un metodo *setup()* che viene eseguito prima dell'esecuzione di ogni test
 - Utile per settare precondizioni comuni a più di un caso di test;
- Un metodo *teardown()* che viene eseguito dopo ogni caso di test
 - Utile per resettare le postcondizioni;
- Un metodo per ogni caso di test.

Struttura di un metodo di test

Inizializzazione precondizioni

Limitatamente alle precondizioni tipiche del singolo caso di test, le altre potrebbero essere nel *setup*.

Inserimento valori di input

Tramite chiamate a metodi set oppure tramite assegnazione di valori ad attributi pubblici.

Codice di test

Esecuzione del metodo da testare con gli eventuali parametri relativi a quel caso di test.

Valutazione delle asserzioni

Controllo di espressioni booleani (*asserzioni*) che devono risultare vere se il test dà esito positivo, ovvero se i dati di uscita e/o le postcondizioni riscontrati sono diversi da quelli attesi.

Scrittura di un caso di test

Scriviamo un metodo *testSomma* che rappresenti un caso di test per il metodo Somma;

Siccome il metodo appartiene ad una classe *calcolatriceTest* nello stesso package della classe da testare, il metodo test può istanziare oggetti della classe ed accedere ai suoi metodi.

```
public void testSomma() {  
    calcolatrice c=new calcolatrice();  
    int a=5,b=7;  
    int s=c.somma(a,b);  
    assertEquals("Somma non corretta!",12,s);  
}
```

Asserzioni

Il metodo *assertEquals* verifica se s (valore ottenuto dall'esecuzione del metodo somma) è uguale a 12 (valore atteso); in caso contrario conta questo fatto come una *failure* e genera il messaggio di errore indicato

```
assertEquals("Somma non corretta!",12,s);
```

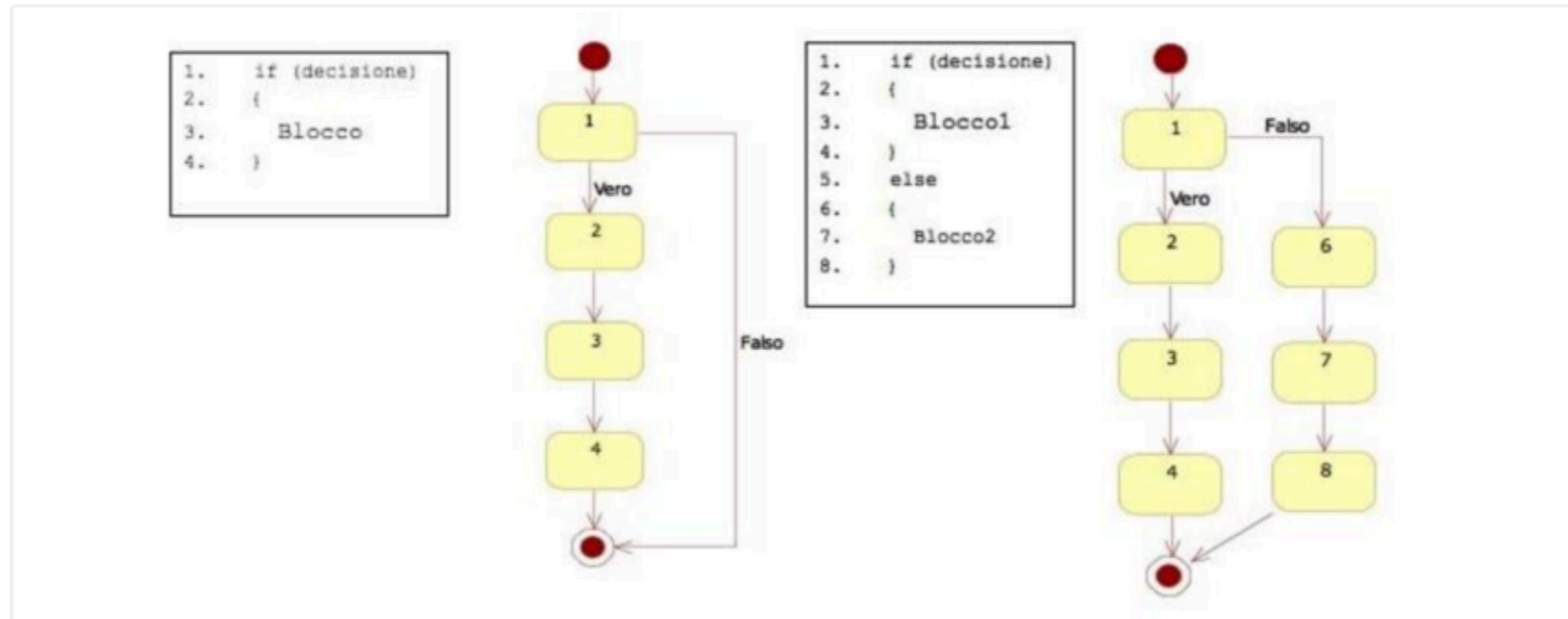
Testing Strutturale (White Box)

Il Testing White Box è un testing strutturale, poichè utilizza la struttura interna del programma per ricavare i dati di test.

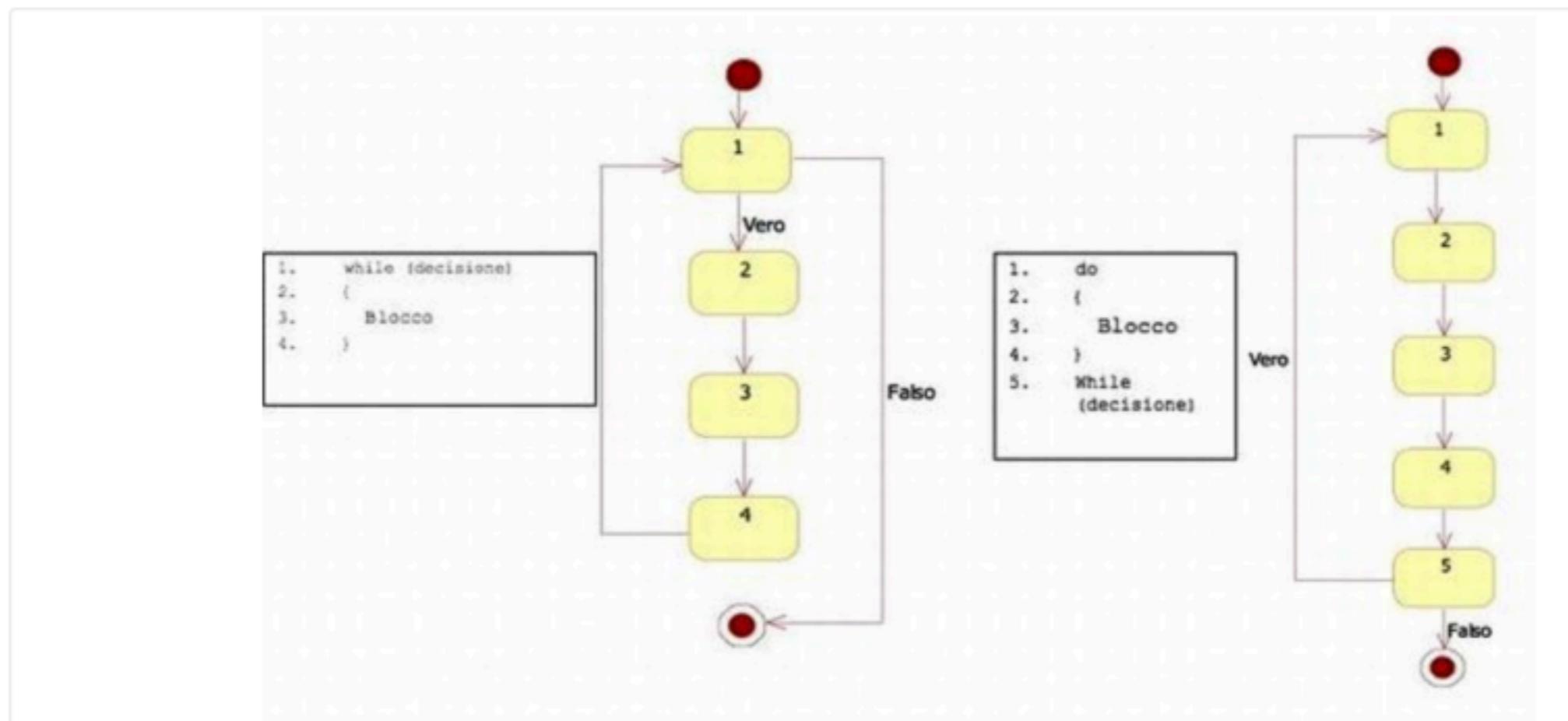
Tramite il testing White Box si possono formulare criteri di copertura più precisi di quelli formulabili con testing Black Box.

Test White Box che hanno successo possono fornire maggiori indicazioni al debugger sulla posizione dell'errore.

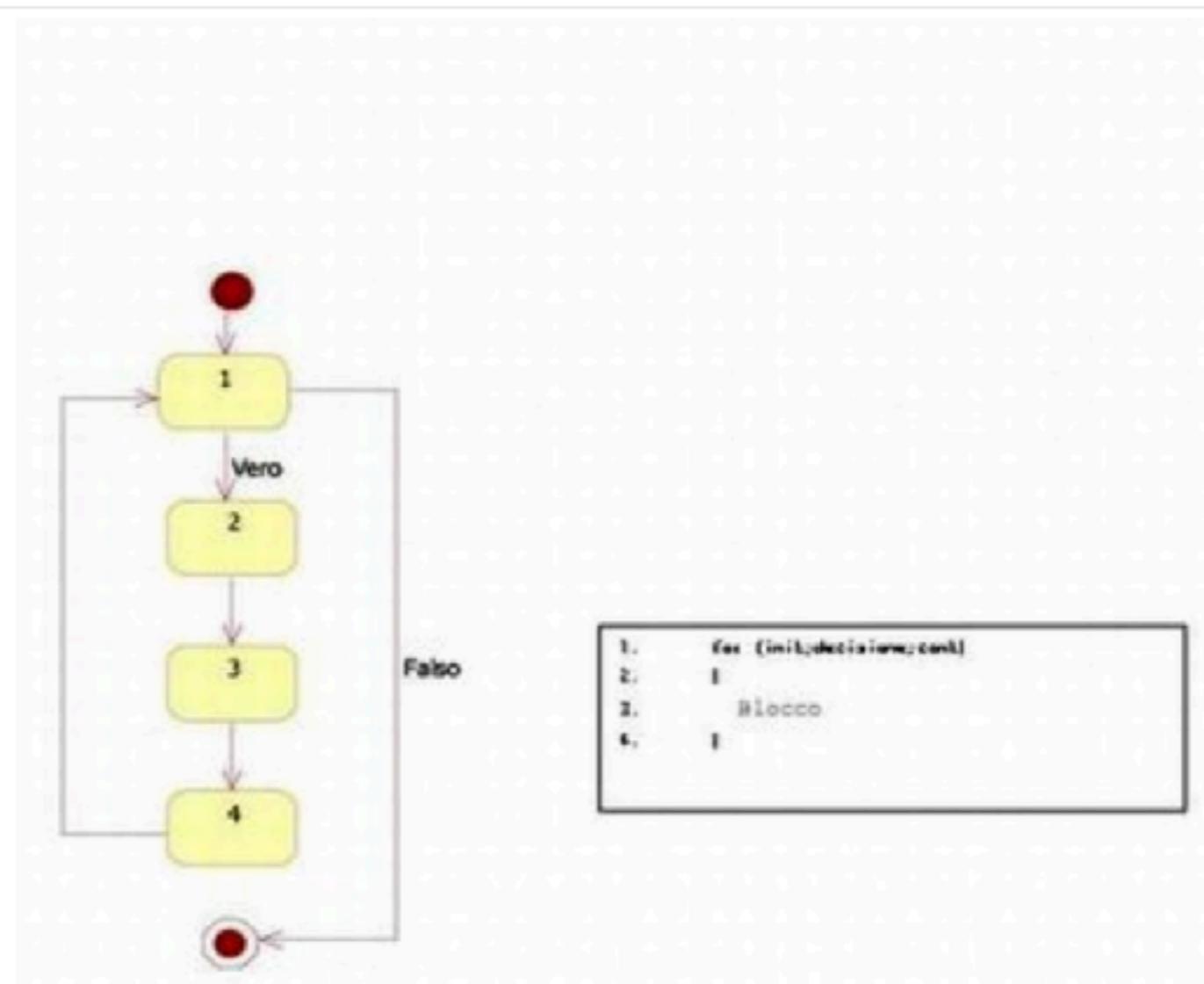
Strutture di controllo e CFG



Strutture di controllo e CFG (segue)



Ciclo For e CFG



Criteri di copertura

Copertura dei comandi (statement test)

- Richiede che ogni nodo del CFG venga eseguito almeno una volta durante il testing;
- è un criterio di copertura debole, che non assicura la copertura sia del ramo true che false di una decisione.

Copertura delle decisioni (branch test).

Richiede che ciascun arco del CFG sia attraversato almeno una volta;

In questo caso ogni decisione è stata sia vera che falsa in almeno un test case.

Copertura delle condizioni (condition test)

Ciascuna condizione nei nodi decisione di un CFG deve essere valutata almeno una volta sia per valori true che false.

Esempio:

```
int check (x); // controlla se un intero è fra 0 e 100
int x;
{ if ((x>=0) && (x<= 200))
check= true;
else check = false;
}
```

TS1={x=5, x=-5 } valuta la decisione sia per valori True che False, ma non le condizioni (la seconda condizione è sempre true).

TS2={x= -3, x=210} è una Test suite che copre tutte le condizioni ma non tutte le decisioni (la and è in entrambi i casi complessivamente false).

Copertura delle combinazioni delle condizioni

Per garantire contemporaneamente sia la copertura di tutte le condizioni che di tutte le decisioni, bisogna cercare di coprire tutte le *combinazioni delle condizioni*.

Es. If ($x > 0 \ \&\& \ y > 0$) ...

TS1={($x = -1, y = -1$), ($x = 2, y = 3$), ($x = 5, y = -4$), ($x = -3, y = 4$)} copre tutte le combinazioni delle condizioni.

Problema:

Nella condizione dell'esempio precedente ($x \geq 0 \ \&\& \ x \leq 200$) non tutte le combinazioni di condizioni sono verificabili (la combinazione *false, false* non è ottenibile).

Cammini linearmente indipendenti (McCabe)

Un cammino è un'esecuzione del modulo dal nodo iniziale del Cfg al nodo finale.

Un cammino si dice **indipendente** (rispetto ad un insieme di cammini) se introduce almeno un nuovo insieme di istruzioni o una nuova condizione **in un CFG un cammino è indipendente se attraversa almeno un arco non ancora percorso**.

L'insieme di tutti i cammini linearmente indipendenti di un programma forma i **cammini di base**; tutti gli altri cammini sono generati da una combinazione lineare di quelli di base.

Dato un programma, l'insieme dei cammini di base non è unico.

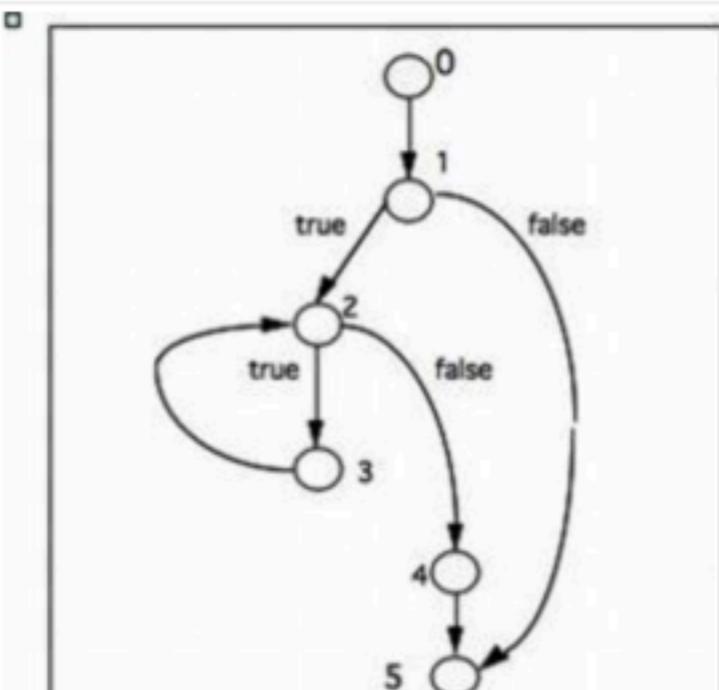
Numero di cammini linearmente indipendenti

Il numero dei cammini linearmente indipendenti di un programma è pari al numero ciclomatico di McCabe:

- $V(G) = E - N + 2$
 - Dove E: n.ro di archi in G – N: n.ro di nodi in G
- $V(G) = P + 1$
 - Dove P: n.ro di predicati in G
- $V(G) = \text{n.ro di regioni chiuse in } G + 1$

Test case esercitanti i cammini di base garantiscono l'esecuzione di ciascuna istruzione almeno una volta e anche il percorrimento di ogni arco.

Esempio



Complessità ciclomatica
del programma è 3

$V(G) = 3 \Rightarrow 3$ cammini indipendenti

$c_1 = 0-1-2-4-5$

$c_2 = 0-1-2-3-2-4-5$

$c_3 = 0-1-5$

Criteri di copertura dei cammini

Copertura dei cammini (path test)

- spesso gli errori si verificano eseguendo cammini che includono particolari sequenze di nodi decisione;
- non tutti i cammini eseguibili in un CFG possono essere eseguiti durante il test (un CFG con loop può avere infiniti cammini eseguibili).

Copertura dei cammini indipendenti

- ci si limita ad eseguire un *insieme di cammini indipendenti* di un CFG, ossia un insieme di cammini in cui nessun cammino è completamente contenuto in un altro dell'insieme, né è la combinazione di altri cammini dell'insieme;
- ciascun cammino dell'insieme presenterà almeno un arco non presente in qualche altro cammino;
- il numero di cammini indipendenti coincide con la complessità ciclomatica del programma.

Relazioni tra i criteri di copertura

La copertura delle decisioni implica la copertura dei nodi.

La copertura delle condizioni non sempre implica la copertura delle decisioni e la copertura dei nodi.

La copertura dei cammini linearmente indipendenti implica la copertura dei nodi e la copertura delle decisioni.

La copertura dei cammini è un test ideale ed implica tutti gli altri.

Esempio

```
1. #include
2. using namespace std;
3. int main ()
4. {
5.     char risposta;
6.     cout<<"\n Vuoi trovare il massimo fra tre interi? (Inserisci N o n per finire)"; cin>> risposta;
7.     while(risposta!='N' && risposta!='n')
8.     {
9.         int num1, num2, num3;
10.        cout<<"\n Inserisci primo numero: "; cin>> num1;

11.        cout<<"\n Inserisci secondo numero: "; cin>> num2;
12.        cout<<"\n Inserisci terzo numero: "; cin>> num3;
13.        int max = num1;
14.        if ( num2 > max )
15.            max = num2;
16.        if ( num3 > max )
17.            max = num3;
18.        cout<<"\n Il massimo e': "<<<"\n";
19.        cout<<"\n Vuoi continuare? (Inserisci N o n per finire)"; cin>> risposta;
20.    }
21.    system("Pause");
22.    return(0);
23. }
```

Testing White box

a) Disegnare il CFG della funzione e calcolarne la complessità ciclomatica;

Progettare un insieme di casi di test (ognuno costituito da un insieme di valori per le variabili in input) in grado di coprire:

b) tutti le istruzioni del programma;

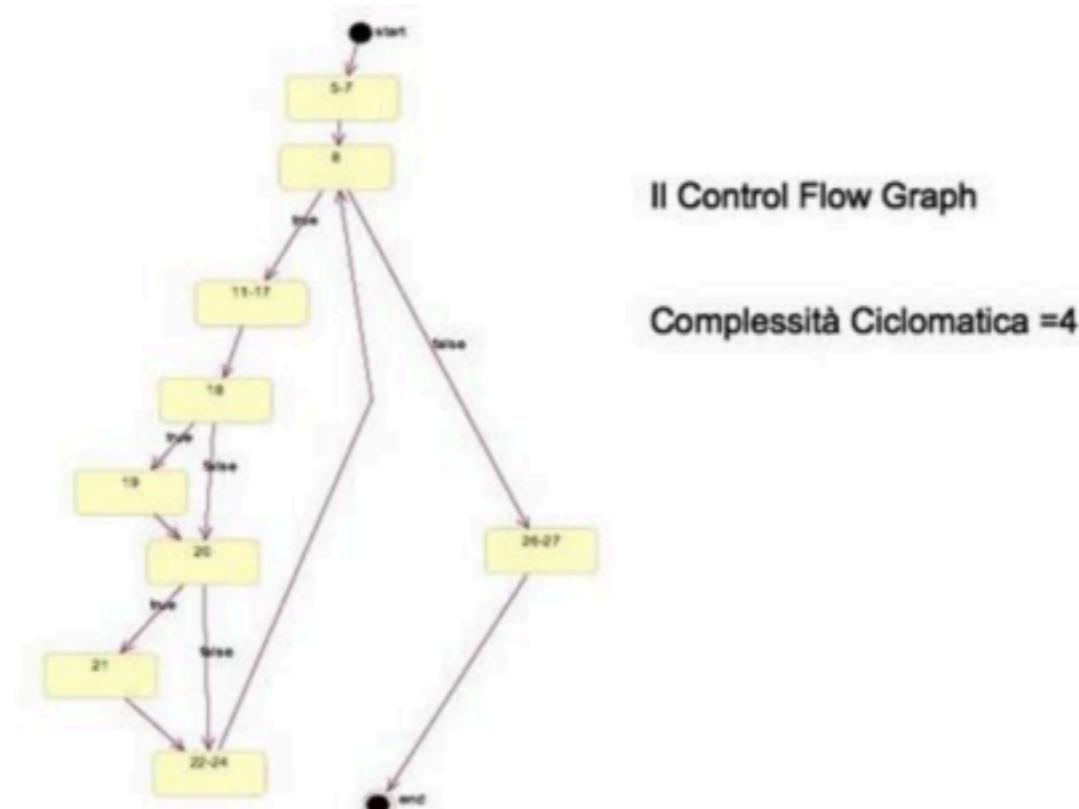
c) tutte le decisioni del programma;

d) tutte le condizioni del programma;

e) tutte le condizioni e decisioni;

f) tutti i cammini linearmente indipendenti.

CFG



I Casi di Test

Copertura	TC	Input (P) Salva	Output Atteso
Istruzioni	TC1	Risposta=S, num1=1, num2=2, num3=3, risposta=N	Max=3
Decisioni	TC2	Risposta=S num1=1, num2=2, num3=3, risposta=s, num1=6, num2=5, num3=4, risposta=n	Max=3 Max=6
Condizioni	TC3	Risposta=S num1=1, num2=2, num3=3, risposta=N	Max=3
Condizioni	TC4	Risposta=n	-
Combinazioni delle condizioni (3/4)	TC3	Risposta=S num1=1, num2=2, num3=3, risposta=N	Max=3 Max=6
Combinazioni delle condizioni (3/4)	TC4	Risposta=n	-

Copertura dei cammini indipendenti

Copertura Cammino	TC	Input	Output Atteso
Start-5-7-8-26-27-28	TC1	Risposta=n	-
Start-5-7-8- 11-17-18- 19-20-21-22-24-8- 26- 27-28	TC2	Risposta=S, num1=3, num2=2, num3=1, risposta=N	Max=3
Start-5-7-8-11-17-18- 19- 20-22 -24-8-26-27- 28	TC3	Risposta=S, num1=1, num2=3, num3=2, risposta=N	Max=3
Start-5-7-8-11-17- 18- 20 -21-22-24-8-26-27- 28	TC4	Risposta=S, num1=1, num2=2, num3=3, risposta=N	Max=3

Livelli di testing

livello produttore

unit testing (Testing di Unità)

integration testing (Testing di Integrazione)

system testing (Testing di Sistema)

livello cooperativo produttore-cliente privilegiato

alpha testing

beta testing

livello cliente o utente

acceptance testing (Testing di accettazione)

Testing di Sistema

Richiede che i vari componenti vengano integrati ed il successivo test del sistema (o sotto-sistema) integrato.

Per sistemi complessi, richiede due fasi:

- **Integration testing** – il team di test accede al codice sorgente. Il sistema viene testato man mano che i vari componenti vengono aggiunti. Punta a trovare i difetti.
- **Release testing** – il team di test testa la versione da rilasciare come una scatola nera (black-box). È più una convalida fatta per dimostrare che il sistema funziona. Se è coinvolto anche il cliente, si parla di Test di Accettazione.

Testing di Integrazione

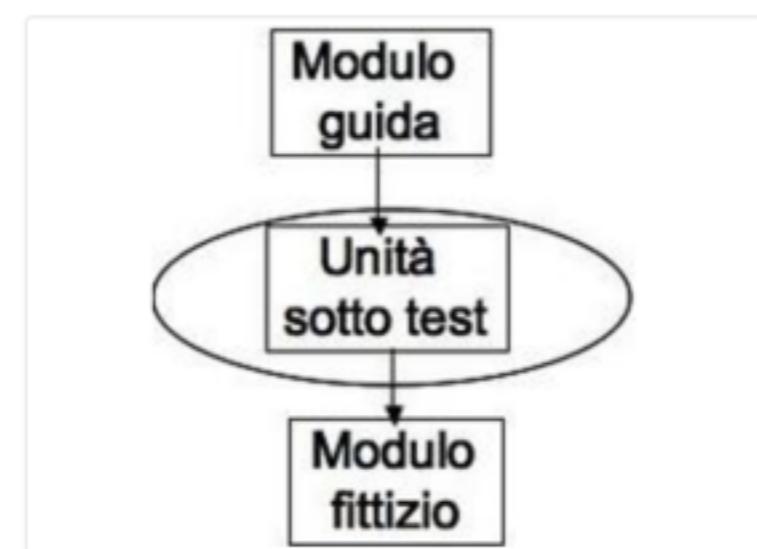
Richiede la costruzione del sistema a partire dai suoi componenti ed il testing del sistema risultante per scoprire problemi che nascono dall'interazione fra i vari componenti.

Richiede l'identificazione di gruppi di componenti che realizzano le varie funzionalità del sistema e la loro integrazione mediante componenti che li fanno lavorare insieme.

Partendo da una architettura organizzata gerarchicamente, le integrazioni possono essere realizzate con approccio **top-down** o **bottom-up** o **misto**.

Esecuzione del test

- Costruzione di Moduli guida (**driver**)
 - invocano l'unità sotto test, inviandole opportuni valori, relativi al test case.
- Moduli fintizi (**stub**)
 - sono invocati dall'unità sotto test;
 - emulano il funzionamento della funzione chiamata rispetto al caso di test richiesto (tenendo conto delle specifiche della funzione chiamata)
 - Quando la funzione chiamata viene realizzata e testata, si sostituisce lo stub con la funzione stessa.



Driver

Tramite un framework di Testing Automation come JUnit realizzare driver e stub necessari per testare un modulo non terminale.
Le classi di test per JUnit rappresentano dei driver quando si riferiscono ad una classe non accessibile dall'esterno;
Un driver deve avere la visibilità dei componenti da testare (o quanto meno dell'interfaccia che vogliamo esercitare).

Stub

Uno stub è una funzione fittizia la cui correttezza è vera per ipotesi.

Esempio, se stiamo testando una *funzione prod_scal(v1,v2)* che richiama una *funzione prodotto(a,b)* ma non abbiamo ancora realizzato tale funzione; nel metodo driver scriviamo il codice per eseguire alcuni casi di test:

Ad esempio chiamiamo *prod_scal([2,4],[4,7])*

Il metodo stub potrà essere scritto così:

```
int prodotto (int a, int b){  
    if (a==2 && b==4) return 8;  
    if (a==4 && b==7) return 28;  
}
```

La correttezza di questo metodo stub è data per ipotesi.

Ovviamente per poter impostare tale testing, bisognerà avere precise informazioni sul comportamento interno richiesto al modulo da testare.

Regression Testing

Man mano che si integrano i vari componenti, sarà anche necessario rieseguire i precedenti test.

Se si scoprono nuovi difetti, si dovrà capire se erano già presenti nei precedenti incrementi, o se sono causati da quelli nuovi.

Il test di regressione si esegue anche dopo un intervento di manutenzione (per verificare che il sistema non sia ‘regredito’).

Confronti fra i vari approcci

- Per la validazione dell'architettura
 - Il Testing di integrazione Top-down scopre meglio gli errori presenti nell'architettura.
- Per la dimostrazione del funzionamento del sistema
 - Il Testing di integrazione Top-down permette una dimostrazione (limitata) fin dalle prime fasi dello sviluppo.
- Implementazione del Test
 - In genere è più semplice con l'approccio bottom-up.
- Osservazione del test
 - Entrambi gli approcci hanno problemi. Può essere necessario codice extra in entrambi i casi.

Testing di Release

È il processo di testing di una release del sistema che sarà rilasciata al cliente.

Obiettivo primario è dimostrare che il sistema si comporti come specificato.

Occorre provare che fornisce tutte le funzionalità, le prestazioni, l'affidabilità, etc. richieste, e che non fallisca.

È in genere un **testing black-box** (a scatola nera) detto anche testing funzionale basato solo sulle specifiche del software;

I Tester non accedono al suo codice.

Performance testing

Dopo che il sistema è stato completamente integrato, è possibile testarne le proprietà emergenti, come le prestazioni ed affidabilità.

I test di prestazione in genere riguardano il carico e si pianificano test in cui il carico viene incrementato progressivamente finché le prestazioni diventano inaccettabili.

Il carico deve essere progettato in modo da rispecchiare le normali condizioni di utilizzo.

Stress testing

Nello stress testing si sollecita il sistema con un carico superiore a quello massimo previsto: in queste condizioni in genere i difetti vengono alla luce.

Stressando il sistema si può testare il comportamento in caso di fallimento.

L'eventuale fallimento dovrebbe essere 'leggero' e non produrre effetti catastrofici. Si deve controllare che non ci siano perdite inaccettabili di servizio e di dati.

Lo Stress testing è particolarmente rilevante per sistemi distribuiti che possono mostrare severe degradazioni delle prestazioni quando la rete è sommersa di richieste.

Piano di test

Documento relativo all'intero progetto.

Struttura

- specifica delle unità di test (per un dato livello di test) Es. Modulo, gruppi di moduli, programma, sottosistemi, intero sistema;
- Caratteristiche da testare: funzionalità, prestazioni, vincoli di progetto, sicurezza...
- Approccio: criterio di selezione dei test, criterio di terminazione, strumenti;
- Prodotti del test: es. Casi di test, rapporto finale, diario del test, statistiche di copertura...
- Schedulazione: quando effettuare il testing e lo sforzo per attività;
- Allocazione del personale.

Rapporti sul test

Diario del test

- descrive i dettagli del test per come si è svolto effettivamente;
- la specifica dei casi di test può essere completata e usata come diario.

Riepilogo del test

- Rivolto al management del progetto
 - numero totale di casi di test eseguiti;
 - numero e tipo di malfunzionamenti osservati;
 - numero e tipo di difetti scoperti.

Sommario dei malfunzionamenti

- Rivolto a chi deve effettuare il debugging o la correzione.

Testing di Classi di oggetti

La completa copertura di una classe richiede di:

- Testare tutte le operazioni di un oggetto;
- Settare ed interrogare tutti gli attributi di un oggetto;
- Esercitare l'oggetto in tutti i possibili stati.

L'ereditarietà rende più difficile la scelta dei casi di test degli oggetti giacchè le informazioni da testare non sono localizzate (ma distribuite nella gerarchia di ereditarietà).

Testing vs Ispezione

Testing e ispezione sono due approcci complementari al problema della verifica del software.

L'Ispezione mira alla verifica della correttezza di un programma, tramite analisi statica e lettura del codice:

- Si basa sull'analisi statica dei programmi;
- Esistono tecniche basati su modelli formali per la valutazione della correttezza degli algoritmi.

Il Testing mira alla **ricerca degli errori** in un programma tramite l'osservazione del suo comportamento e il confronto con il comportamento atteso

- Si basa sull'analisi dinamica dei processi.

L'ispezione è possibile anche per algoritmi che non siano stati ancora implementati, oppure per programmi incompleti.

Il testing è possibile solo per codice che possa essere eseguito.

Sia l'ispezione che il testing possono essere svolte manualmente oppure automatizzate.

Sia l'ispezione che il testing mirano a stabilire l'esistenza di bugs, ma è durante il processo di debugging che si vanno a localizzare e correggere i difetti.

Debugging

Attività di ricerca e correzione dei difetti che sono causa di malfunzionamenti.

Il debugging è ben lungi dall'essere stato formalizzato;

Metodologie e tecniche di debugging rappresentano soprattutto un elemento dell'esperienza del programmatore/tester.

Metodologie per il debugging

"Forza bruta" (brute force)

Si dissemina il codice di sonde per catturare quante più informazioni possibili e valutarli, in cerca di indizi localizzati del malfunzionamento.

Backtracking

Si cerca di ripercorrere il codice "all'indietro" a partire dal punto dove si è verificato il malfunzionamento (istruzione di output oppure eccezione);

Analogamente alla tecnica delle Path Condition diventa via via più difficile procedere all'indietro all'allargarsi del campo di possibilità.

Eliminazione delle cause

Prima di tutto, si individua la tipologia dei dati che fanno fallire il programma.

Poi si cerca di formulare un'ipotesi sulla possibile causa del difetto, proponendo dati in ingresso in grado di far avvenire il malfunzionamento, poi si cerca di controllare la validità di tale ipotesi.

Automatizzazione del debugging

Il debugging è un'attività estremamente intuitiva, che però deve essere operata nell'ambito dell'ambiente di sviluppo e di esecuzione del codice.

Strumenti a supporto del debugging sono quindi convenientemente integrati nelle piattaforme di sviluppo, in modo da poter accedere ai dati del programma, anche durante la sua esecuzione, senza essere invasivi rispetto al codice.

In assenza di ambienti di sviluppo, l'inserimento di codice di debugging invasivo rimane l'unica alternativa.

Funzionalità comuni di debugging

- Inserimento break point.
- Esecuzione passo passo del codice
 - Entrando o meno all'interno dei metodi chiamati;
 - Uscendo.
- Verifica di asserzioni
 - Le asserzioni possono anche essere utilizzate come parametri per break point condizionali.
- Valutazione (watch) del valore delle variabili, mentre l'esecuzione è in stato di interruzione
 - Nei linguaggi interpretati (tra cui anche Java), è possibile anche fornire la possibilità di modificare in fase di esecuzione il valore di variabili, semplificando notevolmente il problema della ricerca di casi di test in grado di replicare il malfunzionamento.