Collaudo ingegneristico

Ogni disciplina ingegneristica fa precedere alla consegna del prodotto il suo collaudo (es. edifici, ponti) in cui:

- Si verifica l'aderenza della realizzazione ai requisiti (un test assicura infinite situazioni corrette)
- Vengono esaminati tutti i documenti di progetto

Il caso del sw

- Convalida (validation): il sw realizza i requisiti del cliente? (Are we building the right product?)
- Verifica (verification): il sw implementa senza errori quanto espresso nella sua specifica? (Are we building the product right?)
  In senso ampio, il termine verifica include la convalida

La verifica nell'ingegneria del sw (differenze rispetto alle altre discipline ingegneristiche)

- Verificare il sw in un punto non garantisce nulla circa il comportamento negli altri

  Esempio `a = …./(x + 32)`

  Ogni valore di `x` va bene eccetto (-32)

- La verifica non è una fase ma uno stile che deve dare forma a tutto il ciclo di sviluppo

- La pianificazione delle attività di verifica è essenziale al fine di:

  ✓Ottenere una visibilità precoce e continua
  ✓Scegliere tecniche appropriate per ciascuno stadio
  ✓Costruire un prodotto testabile

Approcci alla verifica (complementari e non antitetici)

- Testing
  - ✓Tecnica dinamica: si esegue il sw per osservare il suo comportamento
  - ✓Sperimentazione con il comportamento dei prodotti
  - ✓Obiettivo: trovare controesempi
  - ✓Copre la convalida

- Ispezioni
  - ✓Tecnica statica: analisi manuale (o assistita) di codice e documenti
  - ✓Deve precedere il testing del sw
  - ✓Focalizzata sulle proprietà
  - ✓Non copre la convalida

Definizioni (non standardizzate)

- Malfunzionamento (failure): il comportamento del sw è diverso da quello atteso
- Difetto (anomalia, fault, defect, bug): frammento di codice scorretto rispetto alle sue specifiche
- Errore (error): errata interpretazione delle specifiche durante la codifica (o semplice errore di digitazione)

Testing e debugging

1) Testing: cerca di aumentare la probabilità che la presenza di un difetto si manifesti con un malfunzionamento → scopre la presenza di un difetto
2) Debugging: cerca di localizzare il difetto (la cui presenza è stata identificata attraverso il testing) il più in fretta possibile
3) Eliminazione del difetto
4) Ripetizione del test che ha evidenziato il difetto, per accertarsi che sia stato eliminato (per sistemi in tempo reale o concorrenti la ripetibilità può essere difficile)

Testing: una formalizzazione

Il testing non può essere casuale perché inefficace (è cieco, vedi es.
`a = …./(x + 32) )`

- P (programma), D (dominio dei dati d'ingresso), R (dominio dei dati d'uscita)
- Comportamento atteso OK: $D \rightarrow R$ (può essere una funzione parziale)
- Comportamento effettivo di P, assunto in ingresso $d \in D$: P(d)
- P(d) è corretto se $<d, P(d)> \in OK$
- P è corretto se $\forall d \in D, <d, P(d)> \in OK$; si scrive ok(P)

Casi di test

- Test set (o suite): $T \subseteq D$, T finito
- Test case: $t \in T$
- se P(t) è corretto si scrive ok(P,t)
- se, $\forall\ t \in T$, ok(P,t) si scrive ok(P,T)
- T è un test set ideale se $ok(P,T) \Rightarrow ok(P)$
  ovvero, se P non è corretto, esiste almeno un $t \in T$ tale che $<t, P(t)> \notin OK$

Criteri (di selezione) di test

Definizioni

- C è un criterio se $C \subseteq 2^D$
- il test set T soddisfa C se $T \in C$

Proprietà

- C è consistente se ogni test set che soddisfa C fornisce le stesse info: $\forall(T_i, T_j), T_i \in C, T_j \in C, ok(T_i) \Leftrightarrow ok(T_j)$
- C è completo se, nel caso P non sia corretto, esiste almeno un $T_i \in C$ che non ha successo (cioè esiste almeno un $T_i \in C$ per cui $ok(T_i)$ è falso)
- C è completo e consistente: $\forall T \in C$ è un test set ideale $\rightarrow$ consente di provare la correttezza di P attraverso il testing

Tesi di Dijkstra (1972)

“Il testing di un programma può rilevare la presenza di malfunzionamenti ma mai dimostrarne l’assenza”

A sostegno della tesi di Dijkstra, il teorema di Howden afferma che, dato un programma arbitrario P, non esiste alcun algoritmo in grado di derivare un test set ideale (ovvero, non esiste una procedura effettiva per generare un criterio costruttivo che sia consistente e completo)

Testing: una classificazione

- Testing di unità (o di componente o in piccolo): verifica di moduli individuali; di solito (se il sistema non è critico) è di responsabilità dello sviluppatore
  - ✓White box (o strutturale): basato sul codice del modulo considerato
  - ✓Black box (o funzionale): basato sulle specifiche del modulo considerato

- Testing in grande: verifica black box di più moduli o dell'intero sistema, di responsabilità di una squadra indipendente
  - ✓Di integrazione
  - ✓Di sistema
  - ✓Di accettazione
  - ✓Di regressione

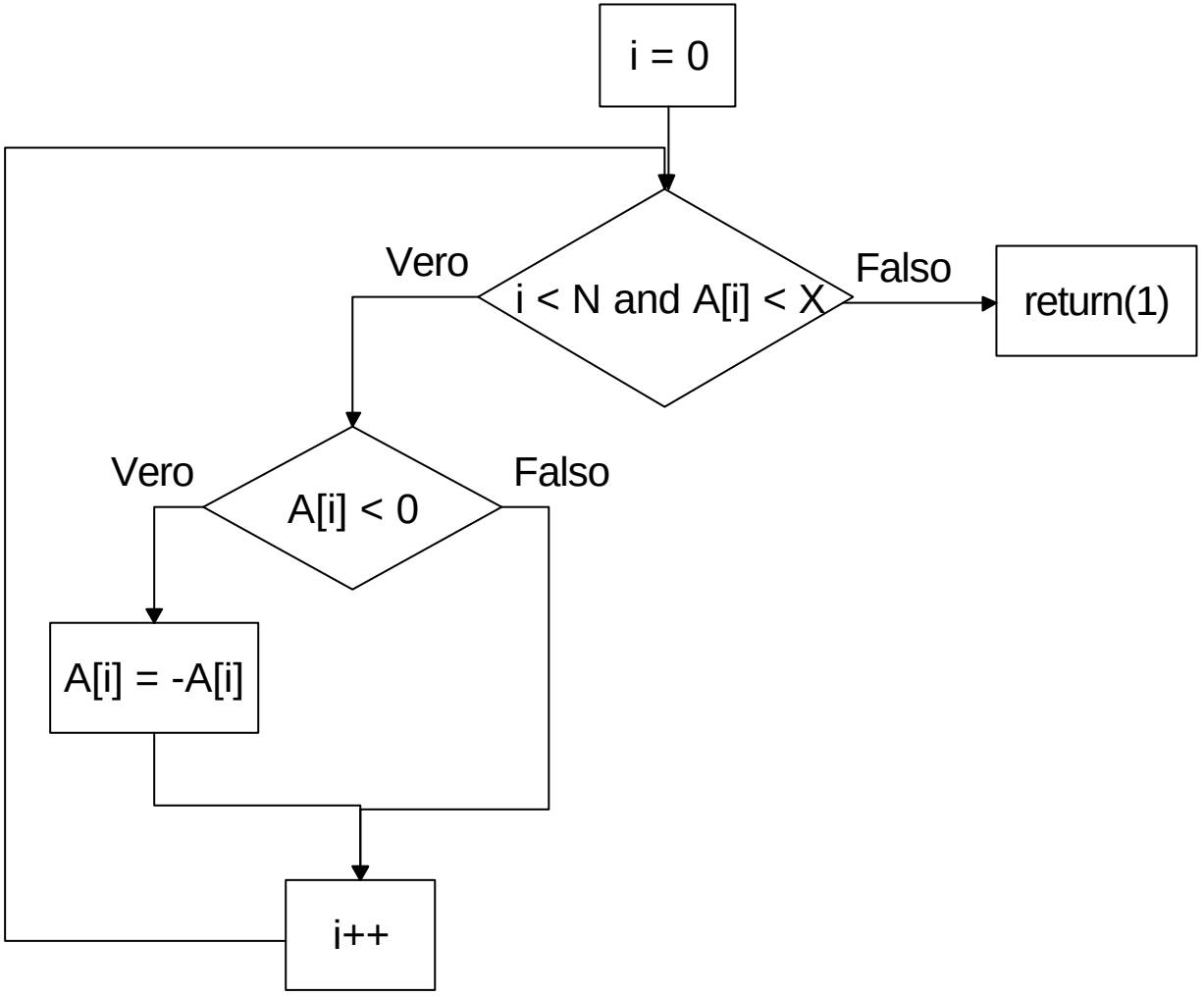
Testing white box (testing di copertura strutturale)

- Deriva i casi di test dal codice del modulo considerato
- È inadeguato se parti significative della struttura del programma non vengono testate
- Criteri di copertura (per cui esistono strumenti automatici di supporto alla generazione dei casi di test)
  - ✓Copertura delle istruzioni (Statement coverage)
  - ✓Copertura dei rami (Branch coverage o edge coverage)
  - ✓Copertura delle condizioni (Condition coverage)
  - ✓Copertura dei cammini (Path coverage)
  - ✓Data Flow (syntactic dependency) coverage

Il modulo M come esempio

```
int trasforma_vettore(int A[], int N, int X)
{
    int i = 0;
    while (i < N and A[i] < X)
    {
        if (A[i] < 0)
            A[i] = - A[i];
        i++;
    }
    return(1);
}
```

Il modulo M come esempio (cont.)



Copertura delle istruzioni

Selezionare un test set tale che ogni istruzione sia eseguita almeno una volta

Per M basta un solo test, ad es. (N=1, A[0]= -7, X=9) a garantire questa copertura

Attenzione: possono esistere istruzioni irraggiungibili

Copertura dei rami

Selezionare un test set tale che ogni ramo del flusso di controllo sia attraversato almeno una volta

Per M basta aggiungere un solo test, ad es. (N=1, A[0]=7, X=9), a garantire questa copertura

Attenzione: possono esistere rami irraggiungibili

Copertura delle condizioni

Selezionare un test set tale che ogni costituente di una condizione composta assuma almeno una volta il valore Vero e almeno una volta il valore Falso

Per M basta aggiungere un test, che causi l'uscita dal ciclo while perché (A[i] < X) è falsa, a garantire questa copertura

Attenzione: possono esistere condizioni irraggiungibili

Copertura dei cammini

Selezionare un test set tale che ogni cammino del flusso di controllo che parte dal nodo iniziale e termina nel nodo finale sia attraversato almeno una volta

Attenzione: possono esistere cammini impercorribili

Problema: in presenza di cicli il numero di cammini è in generale indefinito
Due soluzioni alternative:

- si scelgono solo i cammini linearmente indipendenti (il loro numero è pari al numero ciclomatico e, N.B., non è detto siano tutti cammini che terminano nel nodo finale), cioè si seleziona un test case tale che ogni cammino indipendente sia eseguito almeno una volta
- si limita il numero massimo di percorrenze dei cicli a k (k-limiting), cioè si seleziona un test case tale che ogni cammino contenente un numero di iterazioni di ogni ciclo non superiore a k sia eseguito almeno una volta (k può essere una costante oppure può variare da ciclo a ciclo, in base a considerazioni circa la significatività dei test da condurre)

Data Flow coverage

L’analisi di flusso consente di selezionare cammini significativi da esercitare e, di conseguenza, dato un diagramma di flusso contenente dei cicli, di stabilire il numero massimo di iterazioni di ciascun ciclo che è significativo esaminare (un numero maggiore di iterazioni non permette di esercitare sequenze di definizione-uso delle variabili che non siano già state esaminate con un numero minore di iterazioni)

Copertura dei cammini definizione-uso

1) Identificare i nodi in cui ciascuna variabile viene definita (ovvero le viene assegnato un nuovo valore)

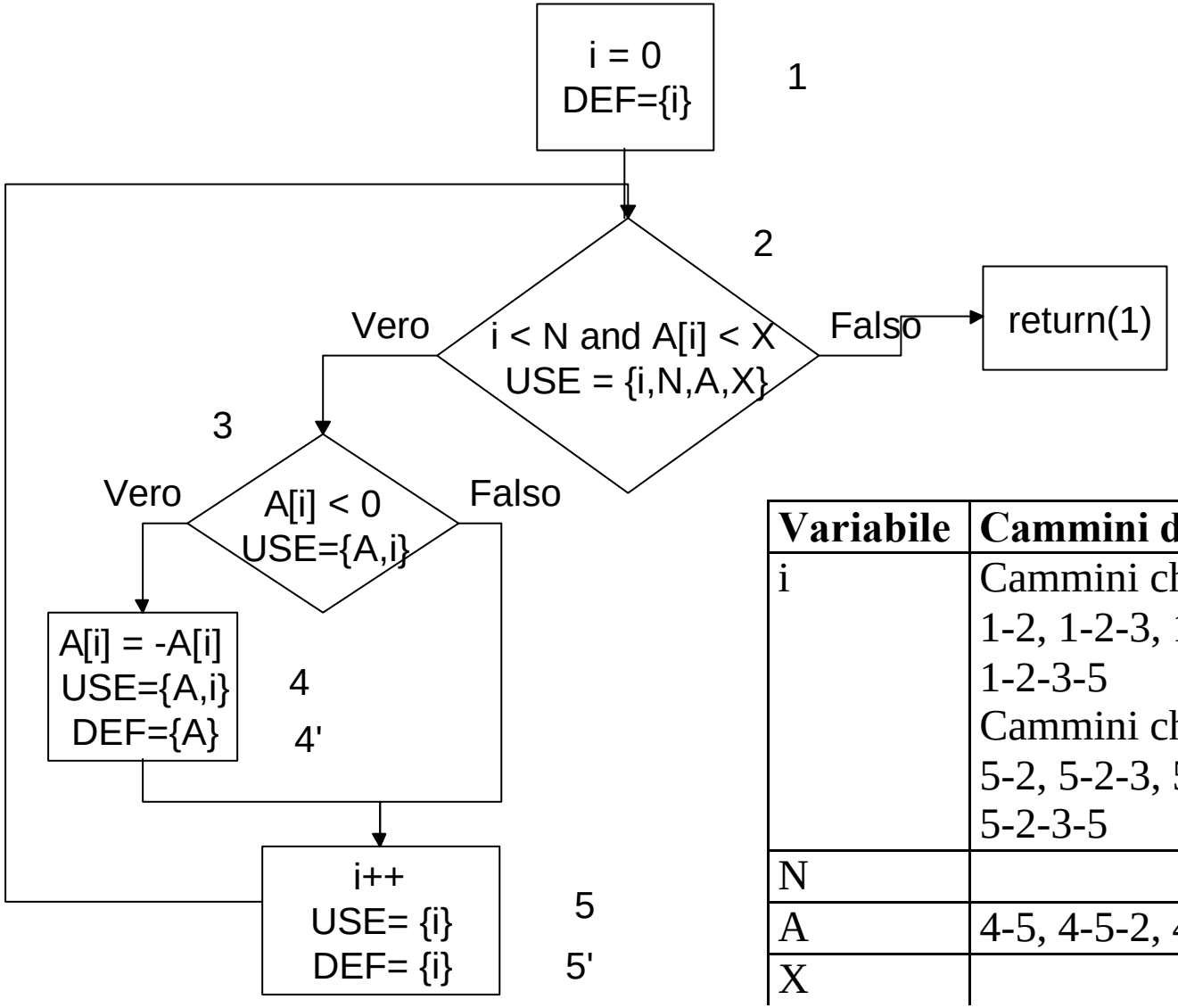
2) Identificare i nodi in cui ciascuna variabile viene usata (ovvero il suo valore viene usato)

3) Identificare tutti i cammini (indipendenti) di definizione-uso di ciascuna variabile

4) Selezionare un test set tale che ogni cammino definizione-uso di ciascuna variabile venga esercitato almeno una volta

Cammino di definizione-uso di una variabile = cammino che parte da un nodo in cui la variabile viene definita, termina in un nodo in cui viene usata e tale che
(a) tutti i nodi intermedi sono liberi da definizioni della variabile considerata (suddividere idealmente ciascun nodo in cui la stessa variabile è sia usata che definita in due nodi in sequenza, il primo in cui viene usata e il secondo in cui viene definita)
(b) non include cicli (liberi da definizioni)

Cammini definizione-uso del modulo M



Variabile	Cammini du
i	Cammini che escono da 1: 1-2, 1-2-3, 1-2-3-4, 1-2-3-4-5, 1-2-3-5 Cammini che escono da 5: 5-2, 5-2-3, 5-2-3-4, 5-2-3-4-5, 5-2-3-5
N	
A	4-5, 4-5-2, 4-5-2-3, 4-5-2-3-4
X	

Valutazione di tutti i testing basati su criteri di copertura

$$\frac{numero\_di\_elementi\_coperti}{numero\_totale\_di\_elementi}$$

Idealmente il rapporto (che, moltiplicato per 100 è espresso in %, ad es. 95% di copertura delle istruzioni) dovrebbe valere 1 ma può essere impossibile ottenere questo risultato

Testing black box

Deriva i casi di test dalle specifiche → se le specifiche sono in linguaggio naturale la generazione dei test set non può essere automatizzata

- Equivalence partitioning
- Boundary value analysis
- Testing guidato dalla sintassi

Equivalence partitioning

Effettuare una partizione del dominio D dei dati d'ingresso del modulo M considerato, $D = D1 \cup D2 \cup \ldots \cup Dn$, dove il comportamento di M in corrispondenza di $\forall t \in Di$ sia simile, ovvero i sottodomini sono classi di equivalenza, dove l'equivalenza è definita empiricamente

Principio della copertura completa
Selezionare un test set che contiene almeno un elemento per ciascun Di

Boundary value analysis (test di frontiera)

Selezionare un test set che, per ciascun sottodominio Di, contenga almeno un valore su ciascun versante della sua frontiera (verso un sottodominio contiguo e/o verso valori che non appartengono al dominio)

Questi test consentono di evidenziare omissioni di controlli (che non possono essere rilevati da test strutturali in quanto questi ultimi verificano solo i rami presenti, non quelli assenti) o di mancato trattamento dei casi di frontiera (ad es. uso di > anziché ≥)

Suggerimento
Aggiungere dei casi di test che rappresentano configurazioni non valide dei dati in ingresso: la probabilità di malfunzionamenti è molto più alta per dati non validi che per dati validi

Esempio: specifiche di una procedura di ricerca

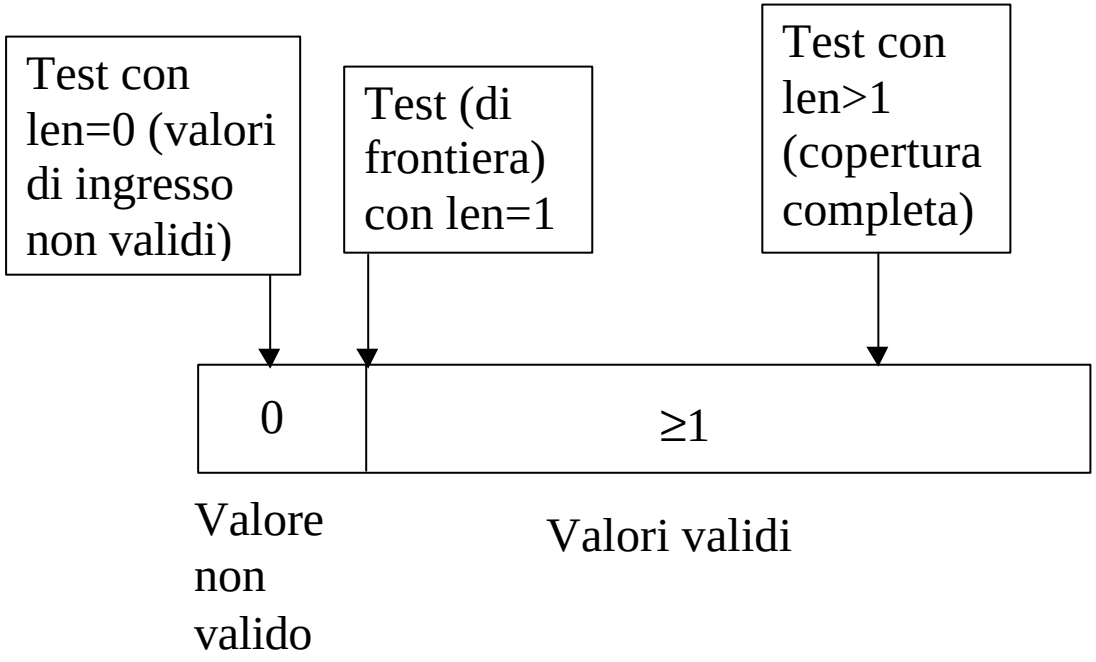
procedure Search (Key : **in** ELEM ; T: **in** ELEM_ARRAY;
Found : **out** BOOLEAN; L: **out** ELEM_INDEX) ;

Pre-condition
-- the array has at least one element
T'FIRST <= T'LAST
Post-condition
-- the element is found and is referenced by L
(Found and T (L) = Key)
or
-- the element is not in the array
(**not** Found **and**
not (**exists** i, T'FIRST <= i <= T'LAST, T (i) = Key))

Esempio: specifiche di una procedura di ricerca (cont.)

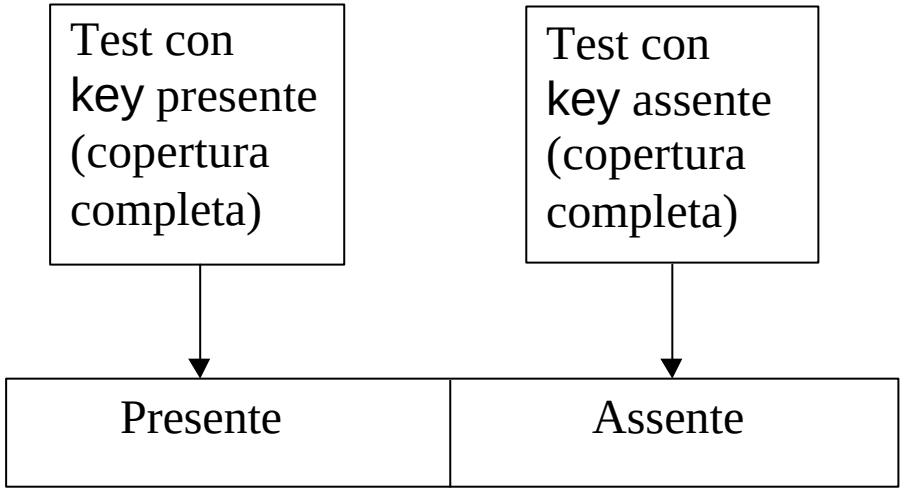
Partizione degli ingressi

Lunghezza dell'array T (len)



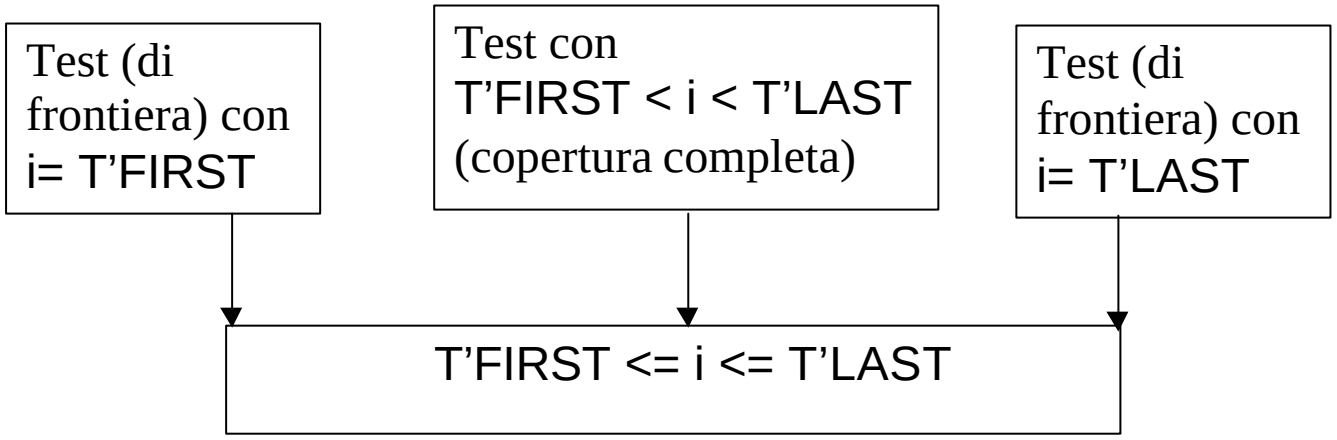
Esempio: specifiche di una procedura di ricerca (cont.)

Presenza dell'elemento key in T



Esempio: specifiche di una procedura di ricerca (cont.)

Posizione i dell'elemento key in T



Esempio: specifiche di una procedura di ricerca (cont.)

Combinazione dei test in un test set

1) Test con array vuoto
2) Test con array contenente un singolo elemento e chiave presente
3) Test con array contenente un singolo elemento e chiave assente
4) Test con array contenente più di un elemento e chiave presente nella prima posizione
5) Test con array contenente più di un elemento e chiave presente in una posizione intermedia
6) Test con array contenente più di un elemento e chiave presente nell'ultima posizione
7) Test con array contenente più di un elemento e chiave assente

Suggerimento: usare array di lunghezza diversa nei test dal 4 al 7

Testing guidato dalla sintassi

Data la sintassi degli ingressi, generare (automaticamente) un test set che copra ciascuna regola sintattica almeno una volta

Es. <expression> ::= <expression> + <term> | <expression> - <term> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= ident | (<expression>)

Testing white box e black box: un confronto

Black box	**White box**
Dipende dalla notazione delle specifiche	È basato sulla copertura del flusso di dati o di controllo
È scalabile	Non è scalabile
Non riesce a distinguere fra implementazioni diverse corrispondenti alla stessa specifica (ad es. ricerca sequenziale o binaria)	Non riesce a rivelare omissioni di cammini (parti delle specifiche che non sono state implementate)

Testing white box e black box: i suggerimenti di Sommerville

Dato un modulo:

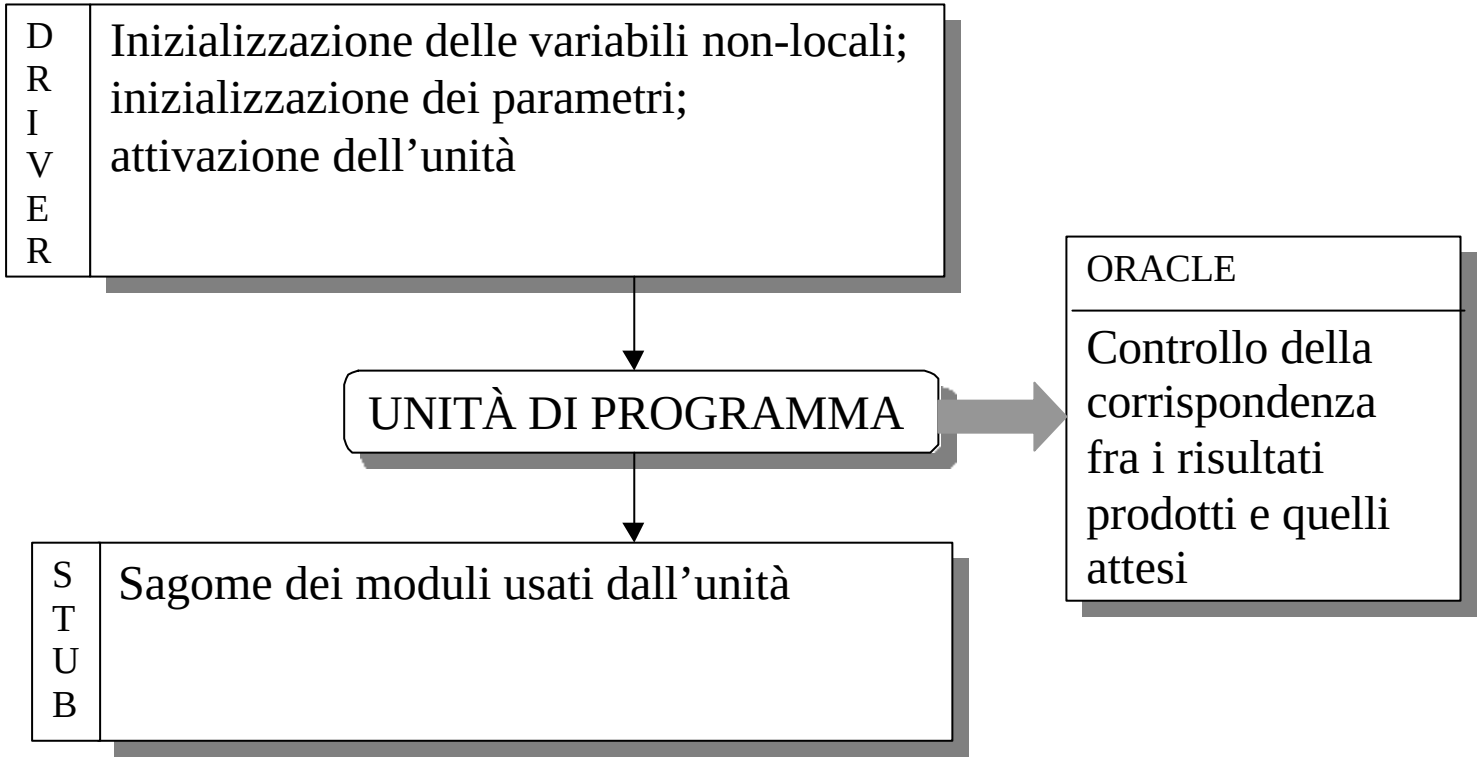
1) Eseguire prima il testing black box
2) Sfruttare la tecnica di equivalence partitioning (applicata ciecamente nel testing black box) per aggiungere nuovi casi di test alla luce della conoscenza del codice
3) Eseguire il testing white box (non esaustivo)

Un’impalcatura per il testing

Sorge il problema di creare un contesto automatico che, dato un test set di unità, sia in grado di

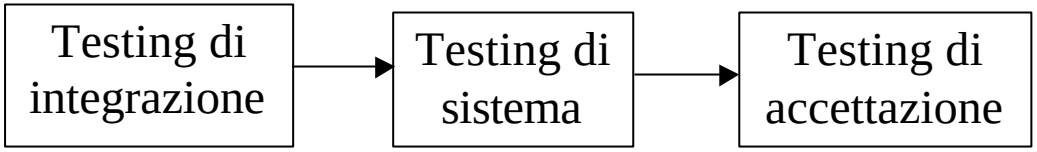
- eseguire separatamente l’unità da testare
- ispezionare i risultati di ciascun singolo test, rivelando gli eventuali malfunzionamenti
- memorizzare i risultati di ciascun singolo test

Un’impalcatura per il testing



Attenzione: è difficile costruire un oracle, non esiste alcuna ricetta universale

Testing in grande



Ricordiamo che tutte queste attività di testing sono black box

Testing di integrazione

- Testing big bang: dopo che tutti i moduli sono stati testati isolatamente, essi vengono integrati tutti insieme e l'intero sistema viene sottoposto a test
- Testing incrementale: i moduli testati isolatamente sono integrati in maniera incrementale e il sistema incompleto viene testato più volte progressivamente usando stub e driver (eventualmente manuali) per sostituire i moduli mancanti

Strategie di testing incrementale

- Top – down: segue l'ordinamento parziale della gerarchia delle chiamate per decidere come procedere all'integrazione; sono necessari stub per i moduli mancanti; è appropriata per scoprire errori architetturali
- Bottom – up: segue in maniera inversa la gerarchia delle chiamate; richiede driver; è appropriata per sistemi OO
- Sandwich: il sistema viene diviso in due layer; il top layer è integrato in maniera top – down, il bottom layer in maniera bottom – up

Testing di sistema

Verifica quei requisiti (soprattutto non funzionali) che si applicano solo al sistema completo; può comprende (fra l'altro):

- Test di stress: test condotti in condizioni di carico (numero di utenti e/o dispositivi) limite (od oltre il limite → il sistema non dovrebbe mai collassare in modo catastrofico)
- Test di sicurezza: test mirati a controllare la riservatezza di servizi e informazioni e la capacità di resistere a tentativi di violazione (accidentali o intenzionali)
- Test di robustezza: test destinati a valutare il comportamento in presenza di dati non corretti
- Test di configurazione: test tesi a evidenziare se il sistema funziona correttamente in ogni configurazione (hw e sw) prevista
- Test temporali: essenziali per i sistemi in tempo reale
- Test di usabilità

Testing di accettazione

Nel caso di prodotto su ordinazione

- può essere previsto dal contratto e il suo esito costituire pregiudiziale per il pagamento
- consente di evidenziare nuovi requisiti o di perfezionare requisiti espressi in maniera vaga o ambigua, per cui costituisce la base per lo sviluppo di una versione successiva del sistema
- consiste in test eseguiti dall'utente finale con dati reali; può trattarsi di
  - ✓ un test set speciale che consente di valutare le prestazioni del sistema (benchmark test): ciò avviene se il cliente ha requisiti particolari e ha commissionato lo stesso sistema a più gruppi → i benchmark test consentono di effettuare la scelta del prodotto da comperare
  - ✓ test che derivano dall'uso quotidiano del prodotto (test pilota) come se fosse installato permanentemente

Testing di accettazione (cont.)

Nel caso di prodotto da distribuire sul mercato, può essere suddiviso in:

- ✓ α - test: il sistema è rilasciato all'interno dell'organizzazione del produttore
- ✓ β - test : un sottogruppo di utenti viene selezionato per la valutazione del sw prodotto

Test di regressione

Verifica che le funzionalità della versione precedente del sistema non siano state corrotte nella nuova versione; al fine di minimizzare lo sforzo:

- della versione precedente salvare
  - ✓l'impalcatura di testing (driver, stub e oracle)
  - ✓i test case
- della nuova versione
  - ✓mantenere traccia dei cambiamenti effettuati
  - ✓valutare l'impatto di questi cambiamenti

Pianificazione e gestione del testing

- Coordinamento delle persone coinvolte
- Scheduling, tenendo conto degli strumenti necessari e delle diverse sedi dove è svolta l'attività
- Specifiche circa la produzione (automatica) di documentazione

Documentazione di testing

- Deve essere oggetto di standardizzazione aziendale
- Il suo contenuto dipende da
  - ✓Caratteristiche dell'organizzazione (dimensioni, avvicendamento, ecc.)
  - ✓Tipo di sw (criticità, vita media, complessità, numero di versioni, ecc.)
- Deve comprendere almeno
  - ✓Documentazione dei test suite eseguiti
  - ✓Documentazione dei test case eseguiti

Documentazione di un test suite eseguito

- Sw testato
- Versione
- Obiettivo (verifica di requisiti e casi d'uso)
- Risultati globali + descrizione
- Autore + data

Documentazione di un test case eseguito

- Obiettivo
- Ambiente (driver, stub, oracle)
- Dati di ingresso
- Uscite attese
- Uscite effettive
- Risultato (Pass/Fail + descrizione)
- Osservazioni (gravità dell'eventuale malfunzionamento, ecc.)

Ispezioni del sw

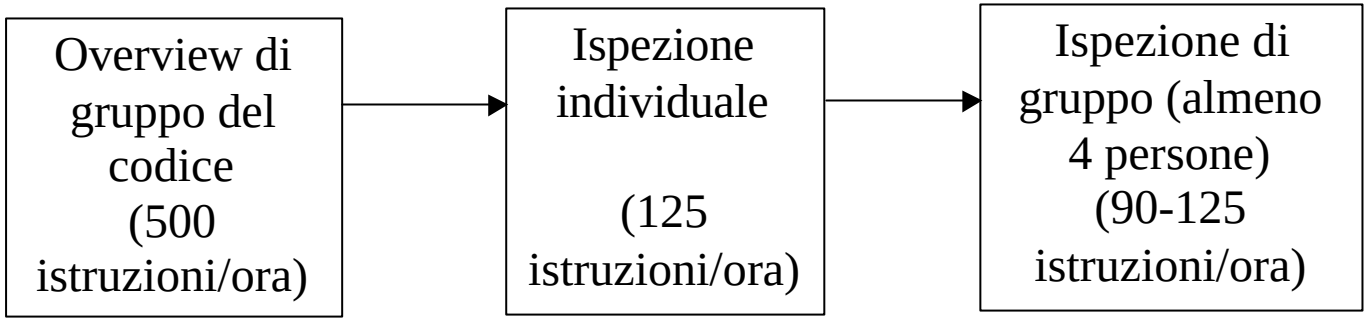
- È una tecnica completamente manuale ma estremamente efficace per scoprire errori (67-85% di tutti gli errori rilevati)
- Può essere applicata nel corso del progetto a tutti gli artefatti, non solo al codice
- Si tratta di analizzare in modo sistematico l'artefatto andando a caccia di difetti catalogati (situazioni erronee o mancanza di conformità con gli standard)
- La checklist dei difetti sw in dotazione a ogni ispettore dipende dal linguaggio di programmazione considerato (tanto più è debole il type checking, tanto più lunga è lista; nel "Software Formal Inspections Guidebook" della NASA, datato 1993, al C sono dedicate 2,5 pagine, al FORTRAN 4)

Fault class	Inspection check
Data faults	Are all program variables initialized before their values are used? Have all constants been named? Should the lower bound of arrays be 0, 1, or something else? Should the upper bound of arrays be equal to the size of the array or size – 1? If character strings are used, is a delimiter explicitly assigned?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statements correctly bracketed? In case statements, are all possible cases accounted for?
Input/output faults	Are all input variables used? Are all output variables assigned a value before they are output?
Interface faults	Do all function and procedure calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer required?
Exception mngt faults	Have all possible error conditions been taken into account?

Checklist: l'esempio NASA

- Suddivisa in: Functionality, Data Usage, Control, Linkage, Computation, Maintenance, Clarity
- Es.:
  - ✓Does each module have a single function?
  - ✓Does the code match the Detailed Design?
  - ✓Are all constant names upper case?
  - ✓Are pointers not typecast (except assignment of NULL)?
  - ✓Are nested "INCLUDE" files avoided?
  - ✓Are non-standard usages isolated in subroutines and well documented?
  - ✓Are there sufficient comments to understand the code?

Ispezioni del codice secondo Fagan (IBM 1976, 1986)



- L'ispezione è un processo costoso
- L'ispezione di gruppo avviene alla presenza dell'autore, che può solo rispondere a domande, se interrogato
- I difetti trovati nelle ispezioni non devono concorrere al giudizio del programmatore ($\rightarrow$ il programmatore non è incentivato a nasconderli) mentre quelli trovati nel testing successivo sì
- Attenzione: durante le ispezioni si devono solo localizzare gli errori, non correggerli (un moderatore ha la responsabilità di fare osservare questa disposizione)

I walkthrough

Differiscono dalle ispezioni perché il codice e la documentazione di accompagnamento vengono pubblicamente presentati ed esaminati dall'autore stesso, che conduce e controlla la discussione, in un'atmosfera più informale

Ispezioni

- Vantaggi
  - ✓La checklist è specifica dell'azienda e viene aggiornata sulla base dei nuovi tipi di difetti riscontrati
  - ✓In un'unica passata si possono scoprire più difetti del codice (anche quelli che si maschererebbero reciprocamente applicando il testing)
  - ✓Si applica anche a programmi incompleti
- Limitazioni
  - ✓Costi elevati
  - ✓La tecnica non è in grado di gestire l'evoluzione del sw

Testing di sistemi OO

I livelli di testing cambiano nel caso di programmazione a oggetti:

- *Method testing*: testing di una singola operazione di una classe
- *Class testing* (unit testing): testing di una classe nella sua globalità
- *Integration testing* (inter-class testing): testing delle interazioni tra più classi
- *Regression testing*: testing eseguito quando una o più classi sono state modificate

Testing di sistemi OO (cont.)

Sono opportuni passi aggiuntivi per rivolgerci specificamente alle caratteristiche OO:

- Quando si aggiunge una nuova sottoclasse o si modifica una sottoclasse esistente, si devono testare i metodi ereditati da ciascuna delle sue superclassi progenitrici
- Quando una sottoclasse sostituisce un metodo ereditato con uno locale omonimo, la sottoclasse deve essere testata con un set test nuovo

Testing di sistemi OO (cont.)

Problema	Spie che devono renderci sospettosi circa la possibile occorrenza del problema
Classe mancante	• Associazioni o generalizzazioni asimmetriche • Attributi e operazioni eterogenei in una classe • Una classe che riveste due o più ruoli • Un'operazione che non ha una classe destinataria • Due associazioni con lo stesso nome e proposito
Classe non necessaria	La classe non ha attributi, operazioni o associazioni
Associazione non necessaria	L'associazione è caratterizzata da info ridondante o nessuna operazione usa l'associazione
Collocazione sbagliata di un attributo	Si deve accedere a un oggetto attraverso un suo valore di attributo

Testing di sistemi procedurali e sistemi OO: differenze

Nella programmazione OO:

- Il test di unità è meno difficile perché gli oggetti sono piccoli
- Il test di integrazione è molto più gravoso perché, nei sistemi OO, la complessità è sospinta verso le interfacce fra componenti
- Per la stessa ragione, le misure di copertura del codice e gli strumenti che le supportano sono di minore valore
- Validare la corrispondenza fra oggetti e metodi, da una parte, e i requisiti del documento dei requisiti, dall'altra, è gravoso perché esistono pochi strumenti (tool) in grado di farlo
- Esistono pochi strumenti in grado di assistere nella generazione di casi di test
- La maggior parte delle metriche del codice (ad es. il numero ciclomatico) sono inutili perché definite per il codice procedurale

Testing di sistemi procedurali e sistemi OO: differenze (cont.)

Sistemi procedurali	Sistemi OO
La componente base è la procedura e un test è fondato su input/output	La componente base è la classe e un test dipende dallo stato della classe → è possibile forzare la classe nello stato desiderato prima di iniziare il test • mediante funzionalità offerte dal linguaggio (es. friend), oppure • modificando il codice sorgente, oppure • si utilizzano le specifiche per determinare le sequenze di operazioni (pubbliche) che portano la classe in un certo stato
Una chiamata di procedura è risolta staticamente (a parte i puntatori a funzione)	Il codice effettivamente eseguito è noto solo run-time (polimorfismo)

Raccomandazioni di Fowler

- Non scrivere codice finché non si ha ben chiaro in mente come verificarlo
- Appena scritto un pezzo di codice, scrivere anche i test relativi
- Non ritenere conclusa la stesura del codice fino a quando tutti i casi di test non sono stati verificati
- Conservare per sempre il codice di verifica ed eseguirlo, ogni volta che è necessario, attraverso una semplice riga di comando o la pressione di un pulsante sull'interfaccia
- Scrivere il codice di verifica in modo che mostri l'eventuale lista dei malfunzionamenti rilevati, già interpretati
- Eseguire test sia per le singole unità (scritti dagli sviluppatori e organizzati in package per verificare le interfacce di tutte le classi), sia per le funzionalità (scritti da un gruppo separato dedicato al testing che guarda al sistema come a una "scatola nera")