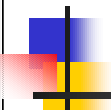


# Tecniche di testing

---

II parte

1



# Testing di software OO

---

2



## Impatto delle Caratteristiche dei Linguaggi OO sul Testing

---

- .... astrazione sui dati, ereditarietà, polimorfismo, binding dinamico, genericità, ....
- Nuovi livelli di test
  - Il concetto di classe come dati + operazioni cambia il concetto di unità
  - Il test di integrazione di oggetti è diverso dal test di integrazione tradizionale
- Nuova infrastruttura
  - Driver e stub devono considerare lo stato (information hiding)
  - Lo stato non può essere ispezionato con tecniche tradizionali

3



## Impatto delle Caratteristiche dei Linguaggi OO sul Testing

---

- Nuove tecniche di generazione di casi di test e criteri di terminazione che tengano conto di
  - Stato
  - Polimorfismo e binding dinamico
  - Ereditarietà
  - Genericità
  - Eccezioni

4



## I livelli di test

- I livelli tradizionali mal si adattano al caso di linguaggi OO
  - Da cosa è rappresentata l'unità?
  - Cosa è un modulo in un sistema OO?

...esistono diverse scuole di pensiero

- Una possibile suddivisione:
  - **Basic unit testing:** test di una singola operazione di una classe
  - **Unit testing:** test di una classe nella sua globalità
  - **Integration testing:** test delle interazioni tra più classi

5



## Stato e information hiding

- Linguaggi procedurali standard
  - **Componente base:** procedura
  - **Metodo di test:** test della procedura basato su input/output
- Linguaggi orientati a oggetti
  - **Componente base:** Classe = struttura dati + insieme di operazioni
  - Oggetti sono istanze di classi
  - La correttezza non è legata solo all'output, ma anche allo stato, definito dalla struttura dati

6



## Problemi

---

- Qual è il componente base da testare?
- Opacità rende più difficile la costruzione di infrastruttura e oracoli
  - E' sufficiente osservare le relazioni tra input e output?
  - Lo stato di un oggetto può essere inaccessibile
  - Lo stato "privato" può essere osservato solo utilizzando metodi pubblici della classe (e quindi affidandosi a codice sotto test)

7




## Polimorfismo e binding dinamico

---

- Linguaggi procedurali classici
  - Le chiamate a procedura sono associate staticamente al codice corrispondente
- Linguaggi orientati a oggetti
  - Un riferimento (variabile) può denotare oggetti appartenenti a diverse classi in relazione tipo-sottotipo ( polimorfismo), ovvero il tipo dinamico e il tipo statico dell'oggetto possono essere differenti
    - Più implementazioni di una stessa operazione
    - Il codice effettivamente eseguito è identificato a run-time, in base alla classe di appartenenza dell'oggetto ( binding dinamico)

8



## Polimorfismo e binding dinamico: problemi

---

Il test strutturale può diventare non praticabile

- Come definisco la copertura in un'invocazione su un oggetto polimorfo?
- Come creo test per "coprire" tutte le possibili chiamate di un metodo in presenza di binding dinamico?
- Come gestisco i parametri polimorfi?



## Ereditarietà

---

- Linguaggi procedurali classici
  - Il codice è strutturato in procedure (che possono essere contenute in moduli)
  - Una volta eseguito il test di modulo di una procedura, generalmente, non è necessario rieseguirlo (salvo modifiche)
- Linguaggi orientati a oggetti
  - Il codice è strutturato in classi
  - L'ereditarietà è una relazione fondamentale tra classi
  - Nelle relazioni di ereditarietà alcune operazioni restano invariate nella sotto-classe, altre sono ridefinite, altre aggiunte (o eliminate)



## Ereditarietà: problemi

---

- Posso “fidarmi” delle proprietà ereditate?
- È necessario identificare le proprietà che devo ri-testare:
  - Operazioni aggiunte, operazioni ridefinite, operazioni invariate, ma influenzate dal nuovo contesto
- Può essere necessario verificare la compatibilità di comportamento tra metodi omonimi in una relazione classe-sottoclasse
  - Riuso test, test specifici

11



## Genericità

---

- Linguaggi procedurali
  - In generale non è possibile definire moduli generici
- Linguaggi orientati a oggetti
  - I moduli generici sono presenti nella maggior parte dei linguaggi OO
  - La genericità è un concetto chiave per la costruzione di librerie di componenti riusabili

12



## Genericità: problemi

- Le classi parametriche devono essere istanziate per poter essere testate
- Che ipotesi posso e devo fare sui parametri?
- Servono classi "fidate" da utilizzare come parametri
- Quale metodo devo seguire quando faccio il test di un componente generico che riuso?
- Non esistono tecniche o approcci specifici in letteratura

13



## State based testing

- Tecnica per testare se i metodi di una classe interagiscono correttamente tra loro, monitorando i data members della classe;
- La tecnica testa tutti i metodi di una classe, uno per uno, rispetto all'insieme degli stati che un oggetto può assumere: se qualche elemento non cambia lo stato dell'oggetto secondo l'aspettato modo, allora il metodo è dichiarato contenente errori;

14



## State based testing

- Ciascun metodo è invocato in tutti i possibili stati che l'oggetto può assumere e dopo ciascuna invocazione si controlla se lo stato risultante è quello atteso;
- Enfasi sulla interazione tra metodi, cioè se il metodo comunica propriamente con gli altri metodi dell'oggetto attraverso lo stato dell'oggetto.
- Gli stati di un oggetto possono essere innumerevoli
  - Necessità di ridurre lo spazio degli stati da esaminare
  - Valori specifici
  - Gruppi di valori generali (suddivisione in classi di equivalenza)

15



## State based testing: processo

- Per ogni data member della classe da testare, definire i valori speciali ed i gruppi di valori generali per specificare i domini di valori da usare per il testing;
- Determinare scenari di dati per la classe;
- Determinare i metodi (operazioni) per 'settare' e testare i valori di stato;
- Per ogni operazione, determinare quali degli stati formano input validi; tali stati devono essere usati per testare l'operazione;
- Iniziare il testing delle operazioni dal fondo del call graph;
- Testare ogni operazione;
- Per ogni stato in cui un'operazione deve essere testata, determinare i valori significativi che devono essere passati come parametri; l'operazione va testata con ognuno di tali parametri.

16



## Testing incrementale per sottoclassi

---

- Tecnica per testare classi appartenenti ad un gerarchia con ereditarietà;
- Quando una classe eredita da una classe base già testata, il testing della sottoclasse richiede il testing solo di alcuni elementi;
- Un sottoclasse R può essere definita come una classe base P più un modifier M.

17



## Testing incrementale per sottoclassi

---

- Con riferimento alla ereditarietà singola, gli elementi di una sottoclasse possono essere classificati in
  - **New**: elementi non esistenti nella classe base;
  - **Virtual-new**: elementi specifici della sottoclasse con binding dinamico;
  - **Redefined**: elementi ereditati ma la cui definizione è modificata nella sottoclasse;
  - **Virtual-redefined**: elementi ereditati e modificati e con binding dinamico;
  - **Recursive**: elementi ereditati;
  - **Virtual-recursive**: elementi ereditati ma con binding dinamico.
- A seconda del tipo di elemento, questo potrà essere testato, riusando eventualmente test case in modo totale o parziale (con aggiunta di altri specifici).

18



## Il debugging

---

19



## Debugging

---

- Conseguenza del testing:
  - Testing: rilevazione di errori
  - Debugging: localizzazione e rimozione di errori
- Processo non del tutto 'disciplinato' da metodi assestati; ancora soggetto alla 'arte' di chi lo effettua;
- Il test fornisce i sintomi del problema: tra sintomo e causa può esservi una relazione né ovvia né semplice;
- Sta al debugging scoprire questa relazione ed apportare le opportune correzioni;

20



## Debugging

---

- Processo difficile implicante una profonda comprensione del codice;
- Condotto ed eseguito in maniera non corretta può inficiare gravemente la qualità del prodotto;
- Alla fine in un processo di debugging si ha una delle due condizioni:
  - La causa dell'errore è stata trovata e rimossa;
  - La causa non è stata identificata, ma si hanno 'sospetti' su di essa; sono necessari altri opportuni test che validino i sospetti.

21



## Difficoltà nel debugging

---

- Sintomo e causa 'geograficamente' lontani;
- Sintomo che scompare temporaneamente dopo la correzione di un altro errore;
- Sintomo causato non da un errore specifico (es. arrotondamenti);
- Sintomo causato da un errore umano, non facilmente individuabile (es. stessa variabile con 2 nomi diversi ma simili per errata digitazione);
- Sintomo dovuto a temporizzazione e non ad elaborazione;
- Difficoltà nel riprodurre esattamente gli input che hanno determinato il malfunzionamento (es. sistemi real time);
- Sintomo intermittente.

22



## Difficoltà nel debugging

- Importanti anche i fattori psicologici: più aumenta la gravità del malfunzionamento più aumenta la probabilità di introdurre nuovi errori dovuti ad una affrettata correzione effettuata
- Il debugging va effettuato sulla base di
  - Valutazioni sistematiche;
  - Intuizioni;
  - Es. Decomposizione binaria sulla base di ipotesi di lavoro che consentono di individuare i nuovi valori da esaminare.

23



## Debugging: approcci

- 'forza bruta'
- Backtracking
- Processo di eliminazione successiva

24



## 'Forza bruta'

---

- Il più comune e, quasi sempre, meno efficiente;
- Viene fatto il dumping della memoria;
- Viene modificato il codice con l'introduzione di sonde che segnalino i cammini eseguiti, ricompilato il programma, rieseguito ed analizzate le stampe (spesso eccessive) delle sonde. L'eccesso di informazioni prodotte può portare al successo ma con gran spreco di risorse.

25



## Backtracking

---

- Abbastanza comune ed usato con successo per molti programmi;
- Si parte dal punto in cui si è rilevato il sintomo (generalmente un'istruzione di output) e si procede all'indietro sul codice sorgente (seguendo il flusso di controllo) fino al punto individuante la causa;
- Programmi con molte linee di codice possono rendere ingestibile il processo.

26



## Processo di eliminazione successiva

---

- Si basa sull'individuazione/deduzione e decomposizione binaria;
- Si considerano (e organizzano) tutti quei dati che possono essere correlati al sintomo;
- Si effettua un'ipotesi e si utilizzano i dati considerati per validarla o confutarla; oppure si costruisce una lista di tutte le cause possibili e test per eliminare le ipotesi scorrette;
- Si effettua un raffinamento dei dati considerati man mano che le ipotesi convergono sulla causa.

27



## Strumenti per il debugging

---

- Compilatori con debugger;
- Tracer;
- Generatori di test case;
- Dumper di memoria;
- Cross reference lister;
- Ma soprattutto la documentazione del sw (completa, aggiornata, consistente, coerente con il codice eseguito).

28



## Un metodo

---

### definire

Cosa                      Quando                      Dove    Fin dove

### uso di check list

#### Domanda

E'

Non è

Cosa è accaduto

Quando?

Dove?

In che misura?

29



## Un metodo

---

- Fare delle ipotesi sulla base delle informazioni riportate;
- Ancora, illustrare e discutere con altre persone del team dell'anomalia introvabile.

30



## Slicing

---

- Introdotta da Weiser nel 1981 per indicare un metodo per la decomposizione automatica di programmi, basato sull'analisi del flusso di controllo e del flusso dati;
- I programmatori implicitamente costruiscono fette (slices) di programma durante la fase di debugging per decomporre programmi complessi in frammenti di codice più piccoli e più semplici da comprendere ed analizzare;
- Definito un sottoinsieme dei comportamenti di un programma, un processo di slicing produce la più piccola sequenza di istruzioni del programma che preserva tale sottoinsieme di comportamenti;

31



## Slicing

---

- I comportamenti di interesse sono definiti sotto forma di criteri di slicing. Weiser definisce un criterio di slicing di una componente di tipo procedura  $P$  come una coppia  $\langle \text{sout}, \text{Vout} \rangle$  dove  $\text{sout}$  è un'istruzione in  $P$  e  $\text{Vout}$  è un sottoinsieme delle variabili in  $P$ ;
- Una slice di una componente di tipo procedura  $P$  su un criterio di slicing  $\langle \text{sout}, \text{Vout} \rangle$ , indicata con  $S(\langle \text{sout}, \text{Vout} \rangle)$ , consiste di tutte le istruzioni e i predicati di  $P$  che possono influenzare direttamente o indirettamente i valori che assumono le variabili in  $\text{Vout}$  immediatamente prima dell'esecuzione dell'istruzione  $\text{sout}$ .

32

```

procedure PosAndEven;
  var n, i, k, npos, neven: integer;
  begin
1.  read(n);
2.  if n > 0
    then begin
3.      npos := 0;
4.      neven := 0;
5.      i := 1;
6.      while i <= n do
          begin
7.          read(k);
8.          if k > 0
              then begin
9.                  npos := npos + 1;
10.                 if (n mod 2) = 0
11.                     then neven := neven
12.                         + 1;
          end;
12.         i := i + 1;
          end;
13.         write(npos);
14.         write(neven);
          end;
    end;
  end;

```

### critério di slicing <13, npos>

```

begin
1.  read(n);
2.  if n > 0
    then begin
3.      npos := 0;
5.      i := 1;
6.      while i <= n do
          begin
7.          read(k);
8.          if k > 0
              then begin
9.                  npos := npos + 1;
12.                 i := i + 1;
          end;
          end;
    end;
  end;

```

33

## Altri criteri di slicing

Lo slicing introdotto da Weiser é anche detto backward slicing; altri ne sono stati introdotti:

- **Forward slicing:** tutti gli statement che possono essere influenzati dai valori delle variabili V nel punto P;
- **Transform slice:** (<sout, Vin, Vout>) tutti gli statement che possono influenzare i valori delle variabili in Vout a partire dai valori delle variabili in Vin immediatamente prima dell'esecuzione dell'istruzione sout;

34



## Altri criteri di slicing

- **Execution slicing:** tutti gli statement eseguiti in una esecuzione del programma;
- **Dynamic slicing:** tutti gli statements che influenzano i valori delle variabili in  $V$  nel punto  $P$  in un caso di prova .

35



## Slice ad un nodo su una variabile

- **Definizione:** Una slice  $S(n, x)$  sulla variabile  $x$  al nodo  $n$  è la chiusura transitiva delle catene definizione-uso da  $n$  a  $x$  e delle dipendenze inverse sul controllo su  $n$ ;
- **Proprietà:** La slice  $S(n, x)$  è l'insieme delle istruzioni compilabili che, se eseguite, produrranno al nodo  $n$  lo stesso valore per  $x$  del programma originale.

36



## Slice ad un nodo

- **Definizione:** Una slice  $S(n)$  al nodo  $n$  è la chiusura transitiva delle catene definizione-uso da  $n$  su tutte le variabili usate da  $n$  e delle dipendenze inverse sul controllo su  $n$ ;
- **Proprietà:** La slice  $S(n)$  è l'insieme delle istruzioni compilabili che, se eseguite, produrranno al nodo  $n$  lo stesso valore per tutte le variabili usate in  $n$  nel programma originale.

37



## Slice inclusiva

- **Definizione:** Una slice inclusiva  $S_{inc}(n)$  al nodo  $n$  è la chiusura transitiva delle catene definizione-uso da  $n$  su tutte le variabili usate da  $n$  e delle dipendenze inverse sul controllo su  $n$  (incluso lo stesso nodo  $n$ );
- **Proprietà:** La slice inclusiva  $S_{inc}(n)$  è l'insieme delle istruzioni compilabili che, se eseguite, produrranno, prima dell'esecuzione del nodo  $n$ , lo stesso valore per tutte le variabili usate in  $n$  nel programma originale.

38



## Slice: esempio

```
main()
{
  int c, nl, nw, nc, inword;
  1. inword=NO;
  2. nl=0;
  3. nw=0;
  4. nc=0;
  5. c=getchar();
  6. while(c!=EOF) {
  7.   nc++;
  8.   if(c=='\n')
  9.     nl++;
  10.  if(c==' ' || c=='\n' || c=='\t')
  11.    inword=NO;
  12.  else if (inword==NO) {
  13.    inword=YES;
  14.    nw++; }
  15.  c=getchar(); }
  16. printf("%d \n", nl);
  17. printf("%d \n", nc);
  18. printf("%d \n", nw);
}
```

$S(16, n) = \{2, 5, 6, 8, 9, 15\}$

$S(18) = \{1, 3, 5, 6, 10, 11, 12, 13, 14, 15\}$

$S_{inc}(17) = \{4, 5, 6, 7, 15, 17\}$

39



## Impatto di un nodo per una variabile

- **Definizione:** L'insieme di impatto  $I(n, x)$  di un nodo  $n$  per la variabile  $x$  è la chiusura transitiva delle catene definizione-uso da  $n$  su  $x$ , e delle dipendenze sul controllo da  $n$ ;
- **Proprietà:** L'insieme di impatto  $I(n, x)$  è quell'insieme di istruzioni che risultano impattate da una nuova definizione della variabile  $x$  al nodo  $n$ .

40



## Impatto di un nodo

- **Definizione:** L'insieme di impatto  $I(n)$  di un nodo  $n$  è la chiusura transitiva delle catene definizione-uso da  $n$  per ogni variabile definita in  $n$ , e delle dipendenze sul controllo da  $n$ ;
- **Proprietà:** L'insieme di impatto  $I(n)$  è quell'insieme di istruzioni che risultano impattate da una nuova definizione di una variabile al nodo  $n$ .

41



## Analisi d'impatto: esempio

```
main()  
{  
1. x=1;  
2. a:  
3. if(x==3)  
4.   x=2;  
5. else if(x) {  
6.     x--;  
7.     goto a;}  
8. printf("%d",x);  
}
```

```
I(4, x)={8}  
I(6)={2, 3, 4, 5, 6, 7, 8}
```

42