Software Testing

Tecniche di Testing per
Software Object Oriented

1

Caratteristiche dei Linguaggi OO

- Il testing di sistemi software OO deve tener conto delle seguenti caratteristiche
  - ✓sempre:
    - » Astrazione sui dati (stato e information hiding)
    - » Ereditarietà
    - » Polimorfismo
    - » Binding dinamico
  - ✓spesso:
    - » Genericità
    - » Gestione delle eccezioni
    - » Concorrenza

2

Impatto sul Test

- 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)
- Nuovi oracoli
  - ✓ lo stato non può essere ispezionato con tecniche tradizionali
- Nuove tecniche di generazione dei casi di test

3

Stato e information hiding

- Linguaggi procedurali standard
  - ✓ componente base: procedura
  - ✓ metodo di test: test della procedura basato su input/output
- Linguaggi object-oriented
  - ✓ 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
  - ✓ lo stato "privato" può essere osservato solo utilizzando metodi *pubblici* della classe (e quindi affidandosi a codice sotto test)

4

Metodi per la generazione di casi di test

- Il test del singolo metodo può essere fatto con tecniche tradizionali (metodo=procedura)
- MA:
  - ✓ metodi più semplici di procedure
  - ✓ scaffolding più complicato
  - ✓ (esempio Push: scaffolding = coda)
- ATTENZIONE: con l'ereditarietà lo stesso metodo può venire usato più volte in contesti diversi …
- può convenire fare il test dei metodi durante il test delle classi, ma …
  - ✓ non si può ignorare il test del singolo metodo !

5

Scaffolding

- L'infrastruttura deve
  - ✓ settare opportunamente lo stato per poter eseguire i test (driver)
  - ✓ esaminare lo stato per poter stabilire l'esito dei test (oracoli)
  - ✓ ma lo stato è "privato" …
- Approcci intrusivi:
  - ✓ modificare il codice sorgente
    - » Aggiungere un metodo testdriver alla classe …
  - ✓ usare costrutti del linguaggio (esempio: costrutto friend)

6

Oracolo

- Usare scenari (sequenze di invocazioni) equivalenti
- Esempio Coda
  - ✓ equivalenti
    - » SEQ1 = create,add(5),add(3),delete
    - » SEQ2 = create,add(3)
  - ✓ non equivalenti
    - » SEQ1 = create,add(5),add(3),delete
    - » SEQ2 = create,add(5)
- In assenza di specifiche formali:
  - ✓ sequenze definite dall'utente in base a conoscenza della classe (specifica informale)

7

Stato e generazione dei casi di test

- La tecnica
  - ✓ costruire una macchina a stati finiti:
    - » stati = insieme di stati della classe
    - » transizioni = invocazione di metodi
- percorrere la macchina a stati finiti per derivare casi di test
- Approcci
  - ✓ basato su specifiche
  - ✓ basato su codice

8

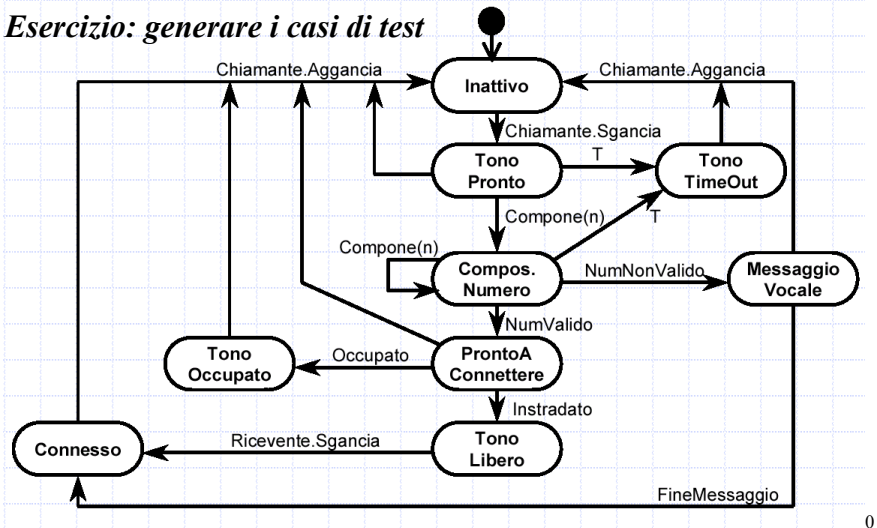
Generazione dei casi di test

- I cammini di una macchina a stati finiti corrispondono a sequenze di esecuzione
- Un insieme di test ragionevole può essere ottenuto selezionando tutti i cammini senza cicli.
  - ✓ Casi di test = cammini della macchina a stati finiti
- Diverse tecniche per
  - ✓ generare macchine a stati finti
  - ✓ limitare numero di cammini
- corrispondono a diversi insiemi di test generati.

9

Esempio: telefonata

Esercizio: generare i casi di test



0

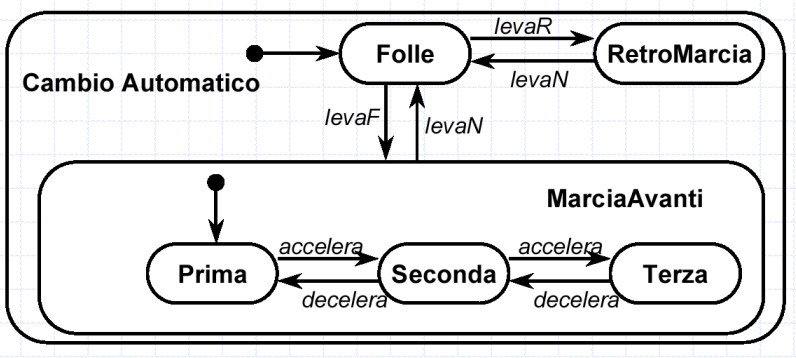
Generazione dei casi di test per una macchina a stati gerarchica

- Stesso principio
- E' necessario ricordare che
  - ✓una transizione in ingresso (uscita) a (da) uno stato decomposto in OR può rappresentare più transizioni elementari che richiedono test distinti
  - ✓uno stato decomposto in AND rappresenta il prodotto cartesiano degli stati componenti. Tutte le possibili coppie devono essere testate.

11

Esempio: cambio automatico

Esercizio: generare i casi di test



12

Criterio di copertura

- La macchina a stati finiti può essere utilizzata per identificare criteri di copertura strutturale:
  - ✓generare sequenze di test a partire da specifiche
  - ✓generare macchina a stati finiti a partire da codice
  - ✓la qualità del test generato (o del software) può essere misurata dal grado di copertura dei cammini della macchina a stati finiti.

13

Problemi rilevabili

- Esistono cammini che non corrispondono a test funzionali:
  - ✓cattiva definizione dei test funzionali (ho dimenticato casi significativi)
  - ✓cattiva specifica (mancano casi significativi)
  - ✓cattiva implementazione (casi non richiesti)
- Esistono test funzionali che non corrispondono a cammini
  - ✓cattiva implementazione (missing paths)

14

Testing di integrazione

- Big bang: in generale poco adatto
- Top-down e bottom-up: cambia il tipo di dipendenze tra "moduli"
  - ✓ Dipendenze: uso di classi ed ereditarietà
    - » Se A usa B allora A dipende da B
    - » Se A eredita da B allora A dipende da B
  - ✓ Preferibile una strategia bottom-up (testare prima le classi indipendenti)
    - » Stub troppo difficili da costruire
- Threads: un thread è identificato con una sequenza di messaggi …

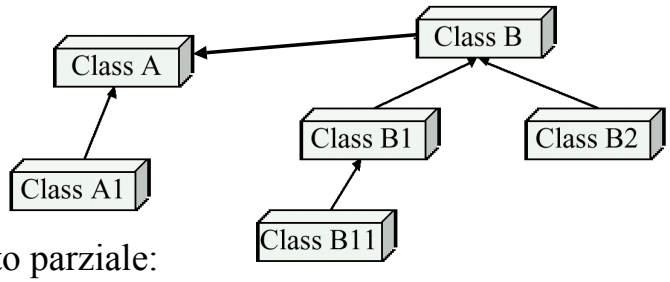
15

Grafo delle dipendenze

- Le dipendenze tra le classi possono essere espresse su un grafo delle dipendenze
- Se il grafo è aciclico esiste un ordinamento parziale sui suoi elementi:
  - ✓ Possibile definire un ordinamento totale topologico
    - » Privilegiare le dipendenze di specializzazione
  - ✓ Ordine d'integrazione definito in base a tale ordinamento
- Se esistono dipendenze cicliche tra le classi è impossibile definire un ordinamento parziale, ma …
  - ✓ Ogni grafo orientato ciclico è riducibile a un grafo aciclico collassando i sottografi massimi fortemente connessi

16

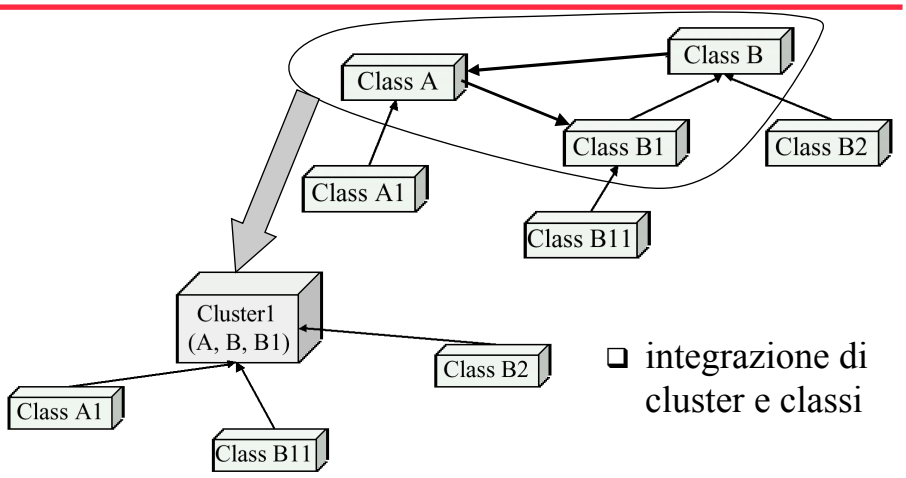
Grafo delle dipendenze aciclico



- ordinamento parziale:
  - ✓ A < A1
  - ✓ A < B
  - ✓ B < B1 < B11
  - ✓ B < B2
- Possibile ordinamento totale: **A < A1** < **B < B2 < B1 < B11**

17

Grafo delle dipendenze ciclico



- integrazione di cluster e classi

impossibile applicare la tecnica sul singolo cluster …

18

Problemi di Integrazione per software OO

- Una volta definito l'ordine di integrazione si aggiungono le classi incrementalmente esercitandone le interazioni
- Possibili problemi di integrazione:
  - ✓ Ereditarietà implica problemi in caso di modifiche di superclassi …
  - ✓ Polimorfismo comporta problemi legati al binding dinamico …

19

Testing di integrazione basato su threads

- La generazione dei casi di test può essere effettuata a partire dai diagrammi di interazione (specifiche)
- Opportuno costruire threads anche dal codice e verificare la corrispondenza con le specifiche …
- Problemi
  - ✓Ereditarietà
  - ✓Polimorfismo e binding dinamico

20

Generazione di casi di test a partire da diagrammi di interazione

- I diagrammi di interazione indicano possibili sequenze di messaggi
- Dovrebbero indicare i casi frequenti e quelli particolari
- Selezione immediata:
  - ✓generare un test per ogni diagramma di interazione
- Selezione più accurata:
  - ✓per ogni diagramma individuare possibili alternative e per ogni alternativa selezionare un ulteriore insieme di casi di test

21

Esempio

Alternative:

- Mario Rossi
  - ✓nome scorretto
  - ✓nome già inserito
  - ✓nome con piano di studi già definitivo
  - ✓…..
- scegli informatica
  - ✓corso diverso da informatica
  - ✓corso inesistente
  - ✓corso non disponibile
  - ✓…..

22

Problemi di Integrazione: 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 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)

23

Ereditarietà: Esempio

```
class Shape {
   public void erase() {…}
   public float area() {…}
   public void
   moveTo(Point p) {…}
   …
   private Point
   referencePoint;
}
```

```
class Circle extends Shape {
      public void erase() {…}
      public float area() {…}
      …
      private int radius;
}
```

- Dobbiamo rifare il test di *moveTo()* nella classe *Circle* ?
- È possibile riusare i test di *Shape* per *Circle* ?
- Come posso controllare la "compatibilità" delle due *erase* e *area*?

24

Ereditarietà: Problemi

- Posso "fidarmi" delle proprietà ereditate?
- È necessario identificare le proprietà che devo ritestare:
  - ✓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

25

Ereditarietà: Possibili soluzioni

- "Appiattimento" delle classi: ritesto ogni metodo ignorandone la gerarchia di appartenenza
  - ✓ vantaggioso per il riuso dei test
  - ✓ altamente ridondante e quindi inefficiente
- Test incrementale
  - ✓ tassonomia dei tipi di proprietà in una classe derivata
  - ✓ storie di test
  - ✓ possibile identificare quali proprietà ritestare, quali test riusare e quali proprietà necessitino di nuovi test
  - ✓ molto più efficiente dell'appiattimento, ma più costosa
  - ✓ basato sul caso pessimo

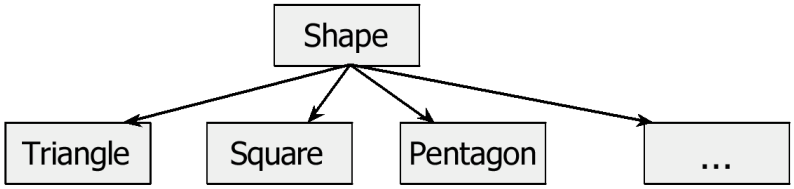
26

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 (polimorfism*o), 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-tim*e, in base alla classe di appartenenza dell'oggetto *(binding dinamic*o)

27

Polimorfismo e binding dinamico: esempio

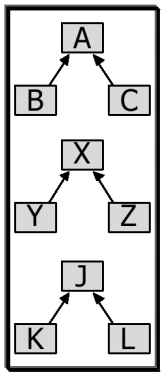


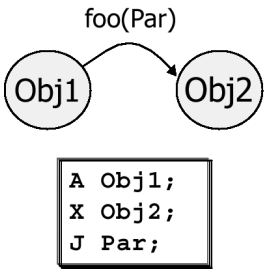
- ***Quale implementazione di area viene effettivamente eseguita?***

```
void foo(Shape polygon) {
     …
     size = polygon.area();
     …
}
```

28

Polimorfismo e binding dinamico: esempio





	Obj1	Obj2	Par
1	A	X	J
2	A	X	K
3	A	X	L
4	A	Y	J
5	A	Y	K
...	...	...	...
27	C	Z	L

Anche in un caso così semplice (singola invocazione di un metodo) ho 27 possibili combinazioni (e non ho tenuto conto dello stato!)

29

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?
- Il test esaustivo può diventare impraticabile
  - ✓ Esplosione combinatoria dei possibili casi
- Definizione di un criterio di selezione specifico
  - ✓ selezione casuale
  - ✓ basata sull'analisi del codice: individuazioni di combinazioni critiche, data flow analysis, …

30

Altri problemi: Genericità

- Le classi parametriche devono essere instanziate per poter essere testate
- Che ipotesi posso e devo fare sui parametri?
- Servono classi "fidate" da utilizzare come parametri (un tipo di *stub* particolare)
- Quale metodo devo seguire quando faccio il test di un componente generico che riuso?

31

Altri problemi: Gestione delle eccezioni

- Le eccezioni modificano il flusso di controllo senza la presenza di un esplicito costrutto di tipo *test and branch*
- Problemi nel calcolare gli indici di copertura della parte di codice relativa alle eccezioni
  - ✓ copertura ottimale: sollevare tutte le possibili eccezioni in tutti i punti del codice in cui è possibile farlo (può non essere praticabile)
  - ✓ copertura minima: sollevare almeno una volta ogni eccezione

32

Altri problemi: Concorrenza

- Problema principale: non-determinismo
  - ✓risultati non-deterministici
  - ✓esecuzione non-deterministica
- Casi di test composti da valori di *input* e *output* sono poco significativi
- Casi di test composti da valori di *input*/*output* e da una sequenza di eventi di sincronizzazione (occorre però forzare lo scheduler a seguire una data sequenza)

33