Verifica e Validazione del Software

Testing Object Oriented

Riferimenti

- Ian Sommerville, Ingegneria del Software, capitoli 22-23-24 (più dettagliato sui processi)
- Pressman, Principi di Ingegneria del Software, 5° edizione, Capitoli 15-16
- Ghezzi, Jazazeri, Mandrioli, Ingegneria del Software, 2° edizione, Capitolo 6 (più dettagliato sulle tecniche)

- **"Object-oriented programming is an exceptionally bad idea that could only have been invented in California."**
  - *Edsger Dijkstra*

Impatto delle caratteristiche OO sul Testing

- Caratteristiche degli oggetti di cui tener conto in fase di testing
  - Stato
  - Ereditarietà
  - Polimorfismo e binding dinamico
  - Genericità
  - Eccezioni
  - Concorrenza

1- Stato ed Information Hiding

- Componente base: Classe = struttura dati + insieme di operazioni
  - oggetti sono istanze di classi
  - la verifica del risultato del test non è legata solo all'output, ma anche allo stato,definito dalla struttura dati

- La 'opacità' dello stato (v. incapsulamento ed information hiding) rende più difficile la costruzione di infrastruttura e oracoli
  - è 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)
    - *In pratica, per testare metodi privati di una classe dobbiamo scrivere metodi di test all'interno della classe stessa*

2- Ereditarietà

- Test ed Ereditarietà
  - 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)

- Ci si può "fidare" delle proprietà ereditate?
  - L'ereditarietà produce una forma di accoppiamento tra la classe figlia e la classe padre, che dovrebbe essere oggetto di testing di integrazione
  - Non è, in generale, corretto riusare i test di un metodo di una classe padre qualora esso sia stato ridefinito da una classe figlia

3- Polimorfismo e Binding Dinamico

- Un riferimento (variabile) può denotare oggetti appartenenti a diverse classi di una gerarchia di ereditarietà (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)

Polimorfismo e problemi per il testing

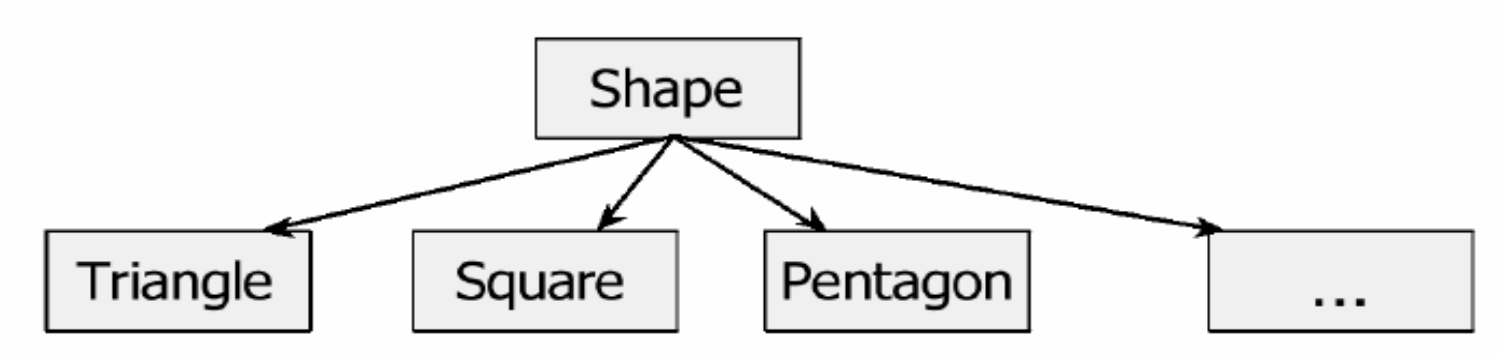
- Il test strutturale può diventare non praticabile:
  - Come definire la copertura in un'invocazione su un oggetto polimorfico?
  - Come creare test per "coprire" tutte le possibili chiamate di un metodo in presenza di binding dinamico?
  - Come gestire i parametri polimorfici?

Problemi della generazione di codice a run-time

- Il problema del binding dinamico rientra nella categoria di problemi legati alla generazione di codice a run-time, comprendente anche:
  - utilizzo di puntatori
  - Utilizzo di puntatori a funzione
  - Generazione dinamica di codice
  - istanziazione dinamica di classi e oggetti (tramite la riflessione) …

- In tutti questi casi, in pratica, è come se non conoscessimo completamente il codice sorgente a tempo di progettazione dei casi di test, quindi non possiamo mai pensare di affrontare perfettamente problemi di copertura
  - Caso estremo: nelle applicazioni Web, il codice lato server genera a tempo di esecuzione delle pagine Web (lato client) comprendenti al loro interno anche codice interpretabile (Javascript)
    - *Non ha senso parlare di copertura nel testing white box del lato client!*

Polimorfismo e Binding Dinamico: esempio




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

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

4- Test e Genericità

- I moduli generici sono presenti nella maggior parte dei linguaggi OO (template in C++, <class> in Java)
  - la genericità è un concetto chiave per la costruzione di librerie di componenti ri-usabili
- Le classi parametriche devono essere instanziate per poter essere testate
  - Bisognerebbe prevedere test per ogni possibile istanziazione della classe parametrica (test esaustivo???)
- Quali ipotesi dover fare sui parametri?
  - Servono classi "fidate" da utilizzare come parametri
- Quale metodologia seguire quando si testa un componente generico che è ri-usato?
  - Non esistono tecniche o approcci maturi in letteratura

5- Gestione delle Eccezioni

- Le eccezioni modificano il flusso di controllo in maniera imprevedibile
  - Agendo come interruzioni, possono causare un salto da una qualsiasi istruzione del try verso le istruzioni del catch
    - *Il CFG, in presenza di gestione delle eccezioni, è inefficace.*

- E' necessario introdurre ulteriori metodi per generare casi di test e ulteriori metriche di copertura per valutare l'effettiva copertura di tutte le 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

6- Concorrenza

- Problema principale: non-determinismo
  - risultati non-deterministici
  - esecuzione non-deterministica

- Casi di test composti solo da valori di Input/Output sono poco significativi
- I casi di test potrebbero essere descritti dai valori di Input/output e dagli istanti di arrivo dei dati in input e dei risultati in output
  - L'istante di arrivo dell'input è però un valore reale, che ammetterebbe un numero illimitato di valori possibili, per cui bisognerebbe trovare delle classi di equivalenza
  - Le classi di equivalenza degli istanti di arrivo individuano condizioni di sincronizzazione tra gli input

Esempio di test di concorrenza

- Nel noto problema dei produttori e dei consumatori nel caso con un solo buffer, 1 produttore P e 1 consumatore C alcuni casi di test significativi potrebbero essere:
  - Precondizione: buffer vuoto
  - TC1: P scrive ‘a’ all’istante t1; C legge a t2>t1. Output atteso: letto ‘a’. Postcondizione: buffer vuoto
  - TC2: C legge a t1; P scrive ‘a’ a t2>t1. Output atteso: nessuna lettura. Postcondizione: ‘a’ nel buffer
  - TC3: P scrive ‘a’ a t1; P scrive ‘b’ a t2>t1. Output atteso: nessuno. Postcondizione: ‘b’ nel buffer
  - …

Utilizzo di Java Path Finder

- Java Path Finder può essere utilizzato per risolvere problemi di concorrenza
- Esso ha, infatti, un motore di analisi del codice in grado di
  - riconoscere possibili problemi di concorrenza
    - *Deadlock, Race conditions, Attese indefinite …*
  - Scatenare combinazioni di chiamate con tempificazioni diverse, allo scopo di ricreare i comportamenti possibili

Java Path Finder Tutorial

- Scaricare Java Path Finder Core e Java Path Finder Symbolic direttamente dal repository
  - http://babelfish.arc.nasa.gov/hg/jpf/jpf-core
  - http://babelfish.arc.nasa.gov/hg/jpf/jpf-symbc
- Attraverso Mercurial
  - https://www.mercurial-scm.org/downloads
- Oppure direttamente dal materiale didattico
- Creare una cartella (windows)
  - C:\Users\<NomeUtente>\.jpf

Java Path Finder Tutorial

- Nella cartella creata scrivere un file site.properties con i percorsi di copia di jpf-core e jpf-symbc:

```
# JPF site configuration
jpf-core = C:/jpf-core
extensions+=,${jpf-core}
# symbc extension
jpf-symbc = C:/jpf-symbc
extensions+=,${jpf-symbc}
```

- Aggiungere ad eclipse il plug-in di Java Path Finder che si trova all'indirizzo:
  - http://babelfish.arc.nasa.gov/trac/jpf/raw-attachment/wiki/projects/eclipse-jpf/update
- A questo punto, in Eclipse, è possibile trovare sul tasto destro di ogni script jpf il comando Verify … che scatena l'esecuzione di Java Path Finder

Esempio: Deadlock tra Filosofi

```
public class DeadlockDetection_Filosofi {

static class Forchetta {}

static class Filosofo extends Thread {

Forchetta forchettaSinistra;
Forchetta forchettaDestra;

public Filosofo(Forchetta forchettaSinistra, Forchetta forchettaDestra) {
this.forchettaSinistra = forchettaSinistra;
this.forchettaDestra = forchettaDestra;
}

@Override
public void run() {
                synchronized (forchettaSinistra) {
                                synchronized (forchettaDestra) {}
                }

}
}
```

```
public static void main(String[] args) {

int nFilosofi = Integer.parseInt(args[0]);

Forchetta[] forchette = new Forchetta[nFilosofi];
for (int i = 0; i < nFilosofi; i++) {
forchette[i] = new Forchetta();
}

for (int i = 0; i < nFilosofi; i++) {

Filosofo p = new Filosofo(forchette[i], forchette[(i + 1)
% nFilosofi]);
p.start();
}
}
}
```

Script jpf

```
target=DeadlockDetection_Filosofi
classpath=${config_path}/bin
search.class = .search.heuristic.BFSHeuristic
# Input per il programma
target.args = 3
```

- search.heuristic.BFSHeuristic rappresenta l'euristica con la quale si cerca di scatenare tutte le combinazioni tra i tempi
- Target.args rappresenta il numero dei filosofi (che in questo caso è il parametro del main)

Output: deadlock trovato

```
====================================================== error 1
gov.nasa.jpf.vm.NotDeadlockedProperty
deadlock encountered:
  thread DeadlockDetection_Filosofi$Filosofo:{id:1,name:Thread-1,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  thread DeadlockDetection_Filosofi$Filosofo:{id:2,name:Thread-2,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  thread DeadlockDetection_Filosofi$Filosofo:{id:3,name:Thread-3,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}


====================================================== snapshot #1
thread DeadlockDetection_Filosofi$Filosofo:{id:1,name:Thread-1,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  owned locks:DeadlockDetection_Filosofi$Forchetta@164
  blocked on: DeadlockDetection_Filosofi$Forchetta@165
  call stack:
at DeadlockDetection_Filosofi$Filosofo.run(DeadlockDetection_Filosofi.java:44)

thread DeadlockDetection_Filosofi$Filosofo:{id:2,name:Thread-2,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  owned locks:DeadlockDetection_Filosofi$Forchetta@165
  blocked on: DeadlockDetection_Filosofi$Forchetta@166
  call stack:
at DeadlockDetection_Filosofi$Filosofo.run(DeadlockDetection_Filosofi.java:44)

thread DeadlockDetection_Filosofi$Filosofo:{id:3,name:Thread-3,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  owned locks:DeadlockDetection_Filosofi$Forchetta@166
  blocked on: DeadlockDetection_Filosofi$Forchetta@164
  call stack:
at DeadlockDetection_Filosofi$Filosofo.run(DeadlockDetection_Filosofi.java:44)
```

Altri esempi

- A disposizione nel materiale didattico:
  - Wait indefinito (causato da accesso errato ad un monitor): DeadlockDetection_Wait
  - Race condition nell'utilizzo di variabili : Race Condition Detection

Appendice