Ingegneria del Software
22b. Altri test e valutazione

Dipartimento di Informatica
Università di Pisa
A.A. 2014/15

criteri *gray-box*

- Una strategia di tipo gray-box prevede di testare il programma conoscendo i requisiti ed avendo una limitata conoscenza della realizzazione, per esempio conoscendo solo l'architettura

- Un'altra strategia gray-box propone di progettare il test usando criteri funzionali e quindi di usare le misure di copertura (si veda la sezione "Valutazione dei test") dei criteri strutturali per valutare l'adeguatezza del test

un approccio (à la Myers)

- Un primo test è progettato utilizzando il grafo causa-effetto
- Il grafo causa-effetto è utilizzato per determinare una partizione del dominio dei dati d'ingresso che sarà usata per integrare il test precedente applicando il criterio dei valori di frontiera
- I progettisti sono chiamati a formulare delle ipotesi di malfunzionamento (*error guessing*) e a integrare di conseguenza i casi di test
- Infine, la struttura del programma è usata per stabilire se i test realizzati ai passi precedenti hanno "esercitato" a sufficienza il codice

esercizio

- Si consideri il seguente programma, che calcola l'importo di una bolletta

```
public double calcolaImporto() {
   double totale = 0.0;
   if (this.consumo <= SOGLIA_CONSUMO_SOCIALE)
      totale = this.consumo * COSTO_UNITARIO_SOCIALE;
      else totale = this.consumo * COSTO_UNITARIO;
   if (!this.esenteIVA)
      totale = totale * (1 + ALIQUOTA_IVA);
   return totale;
}
```

- Si definiscano i valori di input di un insieme di casi di prova per la convalida del metodo, applicando un criterio a scelta di progettazione delle prove

valori delle costanti nell'ambiente di prova

```
private final double ALIQUOTA_IVA = 0.20;
private final double COSTO_UNITARIO = 0.20;
private final double COSTO_UNITARIO_SOCIALE = 0.10;
private final double SOGLIA_CONSUMO_SOCIALE = 50.0;
```

altri test

test di mutazione

- La tecnica si applica in congiunzione con altri criteri di test

- Nella sua formulazione è prevista infatti l'esistenza, oltre al programma da controllare, anche di un insieme di test già realizzati. La strategia prevede di introdurre modifiche controllate nel programma originale

- Le modifiche riguardano in genere l'alterazione del valore delle variabili e la variazione delle condizioni booleane. I programmi così ottenuti, e incorretti – di regola – rispetto alle specifiche, sono definiti *mutanti*. L'insieme dei test realizzati precedentemente viene quindi applicato, senza modifiche, a tutti i mutanti e i risultati confrontati con quelli degli stessi test eseguiti sul programma originale

- Questa strategia è adottata con obiettivi diversi

  - favorire la scoperta di malfunzionamenti ipotizzati: intervenire sul codice può essere più conveniente rispetto alla generazione di casi di test ad hoc

  - valutare l'efficacia dell'insieme di test, controllando se "si accorge" delle modifiche introdotte sul programma originale

  - cercare indicazioni circa la localizzazione dei difetti la cui esistenza è stata denunciata dai test eseguiti sul programma originale

- Uso limitato dal gran numero di mutanti che possono essere definiti, dal costo della loro realizzazione, e soprattutto dal tempo e dalle risorse necessarie a eseguire i test sui mutanti e a confrontare i risultati

test di (non) regressione

- Obiettivo: controllare se, dopo una modifica, il software è regredito, se cioè siano stati introdotti dei difetti non presenti nella versione precedente alla modifica

- Strategia: riapplicare al software modificato i test progettati per la sua versione originale e confrontare i risultati

- Uso in manutenzione. Di fatto, però, il susseguirsi di interventi di manutenzione adattiva e soprattutto perfettiva (e non monotona) rendono la batteria di test obsoleta

- Uso nei processi di sviluppo evolutivi

  - prototipi

  - i test, soprattutto mirati alle funzionalità del prodotto, sono sviluppati insieme al primo prototipo e accompagnano l'evoluzione

  - integrazione top-down

test di interfaccia

- Rivisitazione dei criteri strutturali in termini dell'architettura di un sistema invece che del codice di un programma

- Basati su una classificazione degli errori commessi nella definizione delle interazioni fra i moduli

- *Errore di formato:* i parametri di invocazione o di ritorno di una funzionalità sono sbagliati per numero o per tipo
  - difetto frequente, ma fortunatamente compilatori e *linker* permettono di rilevare automaticamente con controlli statici

- *Errore di contenuto:* i parametri di invocazione o di ritorno di una funzionalità sono sbagliati per valore
  - è il caso in cui i moduli si aspettano argomenti il cui valore deve rispettare ben precisi vincoli; si va da parametri non inizializzati (e.g. puntatori nulli) a strutture dati inutilizzabili (e.g. un vettore non ordinato passato a una procedura di ricerca binaria)

- *Errore di sequenza o di tempo*
  - in questo caso è sbagliata la sequenza con cui è invocata una serie di funzionalità, singolarmente corrette; nei sistemi dipendenti dal tempo possono anche risultare sbagliati gli intervalli temporali trascorsi fra un'invocazione e l'altra o fra un'invocazione e la corrispondente restituzione dei risultati

l’oracolo

- Un metodo per generare risultati attesi…

risultati dalle specifiche

- Risultati ricavati dalle specifiche
  - specifiche formali
  - specifiche eseguibili
    - esempio: grafi causa-effetto
- Inversione delle funzioni
  - quando l’inversa è “più facile”
  - a volte disponibile fra le funzionalità
  - limitazioni per difetti di approssimazione

semplificazione dei dati

- Semplificazione dei dati d'ingresso
  - provare le funzionalità su dati semplici
  - risultati noti o calcolabili con altri mezzi
  - ipotesi di comportamento costante
- Semplificazione dei risultati
  - accontentarsi di risultati plausibili
  - tramite vincoli fra ingressi e uscite
  - tramite invarianti sulle uscite

programmi indipendenti

- Versioni precedenti dello stesso codice
  - disponibili (per funzionalità non modificate)
  - prove di non regressione
- Versioni multiple indipendenti
  - programmi preesistenti (back-to-back)
  - sviluppate ad hoc
  - semplificazione degli algoritmi

valutazione delle prove

- Costi vs. confidenza
  - le prove sono eseguite per sviluppo o per accettazione
  - devono fornire adeguata confidenza
    - la confidenza di una prova costa
- Valutazione di una prova
  - qualità di una prova in sé
  - raggiungimento della confidenza desiderata

la copertura

- Quanto la prova “esercita” il prodotto
  - copertura funzionale: rispetto alla percentuale di funzionalità esercitate
  - copertura strutturale: rispetto alla percentuale di codice esercitata
- Una misura della bontà di una prova
  - la copertura del 100% non significa assenza di difetti
  - non è detto che il 100% di copertura sia raggiungibile

la maturità

- Valutare l’evoluzione del prodotto
  - quanto, in seguito alle prove, il prodotto migliora
  - quanto i malfunzionamenti tendono a “sparire”
  - quanto costa la scoperta del prossimo malfunzionamento
- Definire un modello ideale
  - modello base: il numero di difetti del software è costante
  - modello logaritmico: le modifiche introducono difetti

valutazione delle prove

verifica, validazione e qualità

- Strumento di evidenza
  - a fronte di una metrica e di livelli definiti
  - verificare (validare) per dare evidenza
  - controllo (interno) e assicurazione (esterna) della qualità
- Riferimento: ISO/IEC 25010:2011 (SQuaRE)
  - rimpiazza ISO/IEC 9126
  - quali strumenti per quali caratteristiche?
  - la qualità in uso è esclusa

funzionalità

- Tendenzialmente prove
- Verifica statica come attività preliminare
- Prove per accuratezza

- Liste di controllo (rispetto ai requisiti)
  - appropriatezza (tutte e sole le funzionalità)
  - interoperabilità (soluzioni adottate)
  - sicurezza (soluzioni adottate)
  - aderenza alle prescrizioni

affidabilità

- Tendenzialmente prove
- Verifica statica come attività preliminare
- Prove per maturità

- Liste di controllo (rispetto ai requisiti)
  - tolleranza ai guasti (guasti tollerati)
  - recuperabilità (soluzioni adottate)
  - aderenza alle prescrizioni

usabilità

- Impossibile fare a meno delle prove
- Verifica statica come attività complementare
- Questionari all'utenza (a seguito di prove)
  - operabilità
  - attrazione
- Liste di controllo (rispetto ai manuali)
  - comprensibilità
  - apprendibilità
  - aderenza alle prescrizioni

efficienza

- Impossibile fare a meno delle prove
- Verifica statica come attività preliminare
- Miglioramento vs confidenza
  - l'efficienza deve essere provata
- la verifica statica non dà confidenza, ma migliora il codice
- Liste di controllo (risp. a criteri realizzativi)
  - efficienza algoritmica
  - allocazione/deallocazione delle risorse

manutenibilità

- Verifica statica come strumento ideale
- Prove per la stabilità
- Liste di controllo (batterie di prove)
  - verificabilità
- Liste di controllo (norme di codifica)
  - analizzabilità
  - modificabilità
  - aderenza alle prescrizioni

portabilità

- Verifica statica come strumento ideale
- Prove come strumento complementare
  - installabilità
  - coesistenza
  - rimpiazzabilità
- Liste di controllo (norme di codifica)
  - adattabilità
  - aderenza alle prescrizioni