Verifica – parte IID

Rif. Ghezzi et al.
6.3.5 - 6.3.6

università di ferrara
DA SEICENTO ANNI GUARDIAMO AVANTI.

Test in grande

- Molte delle tecniche viste finora hanno alta complessità, o non sono automatizzabili.
- Possono quindi essere applicate solo a programmi piccoli, o a porzioni di codice.
- Il test di programmi di grandi dimensioni richiede tecniche per affrontare la complessità, esattamente come la progettazione e la specifica.

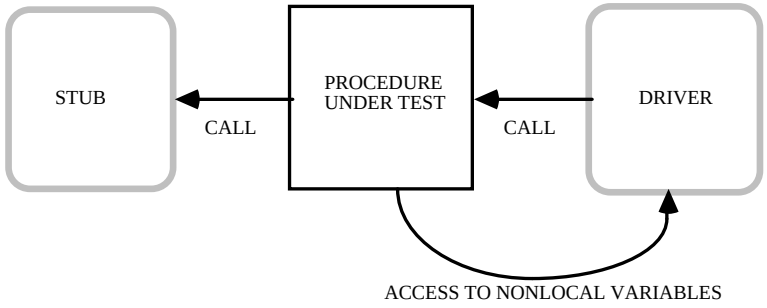
Test e modularità

- La modularità di un progetto guida l'attività di test.
- Attività:
  - Test di modulo (singolo)
  - Test di integrazione: interazioni fra moduli
  - Test di sistema (complessivo)
  - Test di accettazione: eseguito dal cliente

Test di modulo

- Un modulo non può essere testato senza le funzionalità che importa.
- Se un modulo M_i richiede M_h e M_k (Mi USES M_h, M_i USES M_k), e questi non sono disponibili, è necessario simulare le loro risorse.
- Inoltre può essere necessario testare la chiamata di M_i da parte di un altro modulo.
- Scaffolding (ponteggio): ambiente per il test di modulo.

Test di un modulo funzionale



Stub

- Un modulo necessiti di una funzione non ancora sviluppata
- Si può costruire uno stub, ossia una funzione con gli stessi parametri di input e output, ma che simula il calcolo utilizzando un oracolo (come un file o un operatore umano).
- Utilità di specifiche eseguibili (es. logiche)

Driver

- Programma che simula l'invocazione del modulo sotto test.
- Procede all'inizializzazione del modulo sotto esame e alla chiamata di operazioni significative e in sequenze significative.

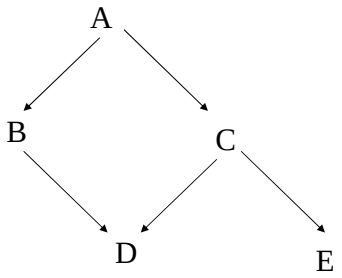
Test di integrazione

- Test big-bang:
  - prima tutti i moduli separatamente
  - poi l'intero sistema
- Si salta la fase di test di integrazione.
- Difetto: i problemi derivati dall'interazione fra moduli si presentano tutti insieme.

Test incrementale

- Vantaggi:
  - si può anticipare il test di integrazione a quando sono pronti i moduli interessati
  - è più facile localizzare gli errori
  - è opportuno testare insieme moduli correlati
  - un'aggregazione parziale di moduli può costituire una fornitura parziale (anche per test di accettazione)
  - può ridurre la necessità di stub e driver

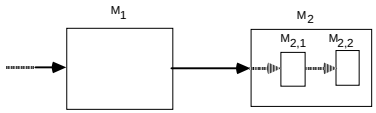
Test bottom-up e top-down: relazione USES



Se la relazione USES è una gerarchia:

- Bottom-up: inizio del test dalle foglie e sviluppo di driver
- Top-down: inizio del test dalla radice e sviluppo di stub.

Esempio: combinazione degli approcci

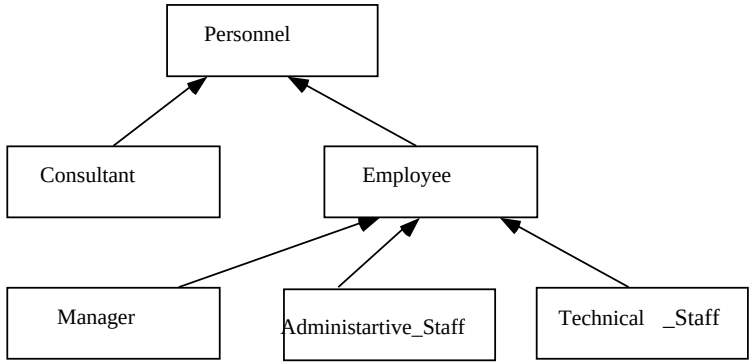


- M_1 USES M_2 e M_2 IS_COMPOSED_OF {$M_{2,1}$, $M_{2,2}$}. Possibilità:
  - Test di M1, con stub per M2 e driver per M1. Poi implementazione di M2,1 e stub per M2,2.
  - Implementazione di M2,2 e test con driver. Implementazione di M2,1 e test di M2 con driver. Implementazione di M1 e test con driver.

Test di software object-oriented

- Problemi specifici
  - Ereditarietà
  - Genericità
  - Polimorfismo
  - Binding dinamico
- Il test di software OO è un argomento di ricerca ancora non assestato.

Ereditarietà



Ereditarietà e test

- L'ereditarietà permette il riuso di codice mediante specializzazione.
- Anche i test, quando possibile, andrebbero riusati.
- In linea di massima sarebbe opportuno:
  - non ripetere il test degli elementi (metodi) invariati rispetto alla superclasse
  - testare i nuovi metodi e quelli ridefiniti
- Possono però esserci interazioni.

Strategie per il test di una gerarchia

- Considerare ogni classe come unità indipendente?
  - Si perde l'incrementalità
- Annotare manualmente i casi di test:
  - Test che non va ripetuto per nessun erede
  - Test che va ripetuto per la classe erede X e tutti i suoi eredi
  - Test che va ripetuto con gli stessi input
  - Test che va ripetuto modificando gli input

Test di codice generico

- Se i test su Table(Int) hanno successo, cosa si può dire dei test su Table(Invoice)?
- Più il codice è generico, più è facile da testare
  - Es: ordinamento basato su operazione generica Precedes anziché <
- Spazio delle possibili istanziazioni dei tipi parametrici:
  - Se è noto, si può procedere esaustivamente
  - Altrimenti, suddivisione in classi (tipi primitivi, strutturati, etc.) e principio di completa copertura

Test di sistema

- Spesso coinvolge componenti non software (hardware, operatori)
- A seconda della struttura del sistema, la modifica di un componente può comportare o meno un nuovo test di sistema
- Tecnica dello scaffolding non limitata al software

Altri tipi di test

- Non solo correttezza funzionale
- Test di tutte le qualità:
  - Test di resistenza al sovraccarico
  - Più in generale, test di robustezza
- Test di regressione
  - Le modifiche al software possono introdurre nuovi errori o manifestarne di già esistenti
  - Tenere traccia dei test su tutte le versioni del software per poterli ripetere in caso di modifiche.

Test di sistemi concorrenti

- Il non-determinismo intrinseco rende difficile la ripetibilità del test

```
task Char_Buffer_Handler is
loop
    select
      when NOT_full accept PUT
      …
    or
      when NOT_empty accept GET
      …
    end select
end loop
end Char_Buffer_Handler;
```


Test di sistemi concorrenti

- Se
  - manca la clausola when NOT_full
  - il buffer è pieno
  - sono pendenti una richiesta di lettura e una di scrittura

  il task può decidere quale servire.
- Se serve prima quella di lettura, il test ha successo
- Se serve prima quella di scrittura, il test fallisce

Test di sistemi real-time

- La correttezza dipende anche dal tempo di esecuzione
- Acquistano quindi importanza fattori che nel caso di programmi non real-time possono essere trascurati
  - hardware su cui si esegue il test
  - software interagente
  - condizioni operative (rete, politica di scheduling…)