test-merged

11. Test e Verifica

Verifica

Non si cercano gli errori di compilazione, ma l'assenza di difetti:

- Si controlla se i risultati sono diversi dalle aspettative
- Si controllano errori di esecuzione, eccezioni, fallimenti

Software senza difetti sono impossibili da avere, serve una continua e attenta verifica su ogni aspetto (specialmente i documenti, design, dati di test, ecc), **anche le verifiche devono essere verificate** e queste verifiche devono essere fatte durante tutto il processo di sviluppo, non solo alla fine.

Bridge Design

Un test assicura infinite situazioni corrette, i programmi non hanno un comportamento continuo e quindi verificare la funzione in un punto solo non ci dice niente circa gli altri punti.

$$a = \ldots / (x + 20)$$

Per ogni valore di x va bene, tranne che per $x = -20$

Per molte qualità le informazioni non sono valori binari (si o no) ma sono soggettive e determinate implicitamente

Approcci

Testing

Consiste nello sperimentare il comportamento del prodotto e fare degli esempi con l'obiettivo di trovare degli errori.

I **test** dovrebbero *identificare la presenza* di errori, che devono essere *localizzati* ed *eliminati* attraverso il **debugging**. Ogni test deve essere ripetuto per vedere se l'errore è stato effettivamente eliminato.

Test case e Test set

- test case: un elemento di D
- test set: un sottoinsieme finito di D (un insieme di test case)

- test set ideale: se P non è corretto allora esiste un elemento in T tale che P è incorretto per quel set
  Il test t ha successo se $P(t)$ è corretto, il set di test T ha successo se P è corretto in ogni $t \in T$

Criteri Di Test

- Un criterio C definisce i test set.
- Un test set soddisfa C se è un elemento di C

Proprietà

- **consistenza**: per ogni coppia (T1, T2) di test soddisfatti da C, T1 ha successo se T2 ha successo, quindi ognuno da la stessa informazione
- **completezza**: se P non è corretto, c'è un test set di C che non ha successo

C è completo e consistente se identifica un test set ideale e permette alla correttezza di essere provata

- C1 è più affidabile di C2 se per ogni programma, per ogni test soddisfatto da C1 c'è un sottoinsieme T2 di T1 che soddisfa C2

Important

Non esiste un algoritmo che genera un test-set che possa provare la correttezza di un programma, non c'è un criterio di costruzione che sia costintente e completo.

Criteri Empirici

Per alcuni programmi andrebbero eseguiti veramente troppi test, per questo si cerca di dividere D in tanti sottodomini D_i dove ogni elemento dovrebbe avere comportamento simile.

Successivamente si seleziona un test per ogni sottodominio, se $D_j \cap D_k \neq 0$ si prende uno degli elementi dell'intersezione per poter ridurre i test.

Moduli Di Test

Ci sono due approcci:

- **black box** o test funzionali: partizionano i criteri in base a delle specifiche senza conoscere i dettagli interni.
- **white box** o **test strutturali**: partizionano i criteri in base al codice interno del modulo, conoscendone la struttura.

Analisi

Studio analitico delle proprietà, è una tecnica statica e formale.

Definizioni

- P (programma)
- D (dominio di input)
- R (dominio di output)

$$P : D \to R$$

Un programma è una funzione che mappa D in R.

Correttezza

$$OR \subseteq D \times R$$

- P(d) è corretto se la coppia $< d, P(d) > \in OR$
- P è corretto se $\forall d \in OR$ tutti i $P(d)$ sono corretti
- FAILURE: può essere indefinito oppure potrebbe essere un risultato errato
- ERROR: qualsiasi cosa che causi un fallimento
- FAULT: stato intermedio sbagliato in cui entra un programma

> **Important**
>
> Queste definizioni non sono standardizzate

12. Test Funzionali

Si basano sul comportamento **ingresso-uscita** che il software presenta nel suo ambiente operativo, le tecniche di progettazione si basano sul **ricavare un certo numero di test case** e permettono *anche* di verificare il *mancato soddisfacimento di requisiti non funzionali*. Sono **complementari** ai test strutturali

Casi Di Test

I casi di test devono essere definiti *prendendo in considerazione* le condizioni che corrispondono a classi di input/output **non valide** e **valide**. Ciascun caso di test deve essere *rappresentativo di una classe* in modo da **minimizzare il numero totale di test** da effettuare

Le **tecniche di test principali** per definire i casi di prova sono:

- La tecnica della copertura delle classi di equivalenza
- La tecnica di **analisi dei valori estremi**
- La tecnica di **copertura delle funzionalità**

Copertura Delle Classi Di Equivalenza

> **Classe di equivalenza**
>
> Un sottoinsieme dei dati in input tale che il test di ogni elemento abbia lo stesso risultato dal punto di vista del comportamento ingresso-uscita

La tecnica prevede 2 passi:

1. **Identificazione** delle classi
2. **Definizione di casi di test** che coprano le classi

A partire dalle **specifiche funzionali** possono essere identificate diverse classi (valide o non valide)

Criteri Utili per L'identificazione Delle Classi

- Intervallo di valori di input
- Numero di valori di input
- Insiemi di valori di input

Intervallo Di Valori Di Input

Se una condizione di ingresso specifica un **intervallo di valori ammissibili** per un determinato parametro di input si identificano:

- Una **classe di equivalenza valida** per i valori compresi nell'intervallo
- Due **classi di equivalenza non valide** per i valori inferiori e superiori all'intervallo

Numero Di Valori Di Input

Se una condizione di ingresso specifica un **numero/quantità di valori ammissibili** per un determinato parametro di input si identificano:

- Una **classe di equivalenza valida** per un numero/quantità compreso fra il minimo ed il massimo specificati
- Due **classi di equivalenza non valide** per un numero di valori inferiori e superiori

Insiemi Di Valori in Input

Se una condizione di ingresso specifica un **insieme di valori ammissibili** per un determinato parametro di input si identificano:

- una **classe di equivalenza valida** per ogni gruppo di elementi dell’insieme che si pensa siano trattati in modo analogo
- una **classe di equivalenza non valida** per un elemento non appartenente all’insieme

Progettare Casi Di Test

A partire dalle classi identificate occorre progettare un numero di casi di test sufficiente a **coprire tutte le classi di equivalenza valide**, facendo in modo che ciascun caso di test copra il maggior numero possibile di classi valide. Bisogna inoltre creare tanti casi di test **quante sono le classi di equivalenza** non valide in modo tale che ciascun caso di test copra una ed una sola classe non valida

Analisi Dei Valori Estremi

Le condizioni sui valori estremi sono quelle condizioni che si trovano direttamente su un valore estremo di una classe di equivalenza di ingresso o di uscita, **immediatamente al di sopra** di esso oppure **immediatamente al di sotto**

- I casi di test che esplorano condizioni su valori estremi del dominio di input sono *molto produttivi*
- Una **generazione casuale** dei casi di test, in generale, **non individuerebbe** la maggior parte di questi difetti

> **Differenze con le classi di equivalenza**
>
> - Sono scelti come rappresentativi della classe di equivalenza uno o più valori in un intorno di ciascun estremo
> - I casi di test sono progettati considerando anche l'output (classi di equivalenza di uscita)

Individuazione Delle Classi

I criteri per l'identificazione delle classi per estremi sono analoghi ai precedenti.

Per ciascun intervallo di valori ammissibili in ingresso ed in uscita occorre progettare:

- casi di test validi sugli estremi dell'intervallo
- casi di test non validi per i valori immediatamente al di sotto del minimo e al di sopra del massimo

Per ciascun numero di valori ammissibili in ingresso ed in uscita progettare

- casi di test validi per il numero minimo e per il massimo
- casi di test non validi per i numeri immediatamente al di sotto del minimo e al di sopra del massimo

Tecnica Di Copertura Delle Funzionalità

Richiede di:

- Analizzare le specifiche al fine di determinare le funzionalità elementari del prodotto indipendenti fra loro
- Progettare casi di test che coprano tutte le funzionalità

Per verificare la completa copertura si definisce una **matrice di test**

BlackBox

Tecniche per la progettazione dei casi di test funzionali input/output

- Copertura delle classi di equivalenza
- Analisi dei valori estremi
- Copertura delle funzionalità

Fasi dell'attività di test

- Progettazione e pianificazione dei casi di test
- Esecuzione del software per ciascun caso di test e registrazione del comportamento del prodotto
- Confronto tra il comportamento atteso e quello reale

13. Test Strutturali

Idea generale dei test strutturali:

- **Criterio di inadeguatezza**: se parti della struttura *non sono testate* il test è *inadeguato*
- **Criterio di copertura del control flow**:
  - Statement coverage
  - Edge coverage
  - Condition coverage
  - Path coverage
  - Data flow coverage

Statement Coverage

Il criterio di **copertura dei comandi** comprende la selezione di un test set tale che ogni comando o **statement** di P *sia eseguito da un qualche test case*. Se ogni D_i è l'insieme di dati che esegue il comando i allora bisogna tentare di **minimizzare** l'insieme dei D_i in modo da **avere una partizione**.

Edge Coverage

Il criterio di **copertura degli archi_** comprende la selezione di un test set tale che ogni arco o **branch** del control flow *sia attraversato almeno una volta* da un qualche test case, bisogna ovviamente minimizzare la dimensione del test set.

Condition Coverage

Il criterio **delle condizioni composte** comprende la selezione di un test set tale che *tutti i possibili valori costituenti le condizioni composte siano testate* almeno una volta.

> ⚠ **Attention**
>
> Considera tutti i possibili modi di rendere vera o falsa una condizione composta, **solo se** la condizione non è composta corrisponde all' Edge Coverage.

> ⚠ **Attention**
>
> Se la condizione ha n componenti booleani si possono avere sino a 2^n possibili assegnazioni ai componenti

Path Coverage

Il criterio di **copertura dei cammini** comprende la selezione di un test set che *attraversa tutti i cammini dal nodo iniziale al nodo finale* del diagramma di flusso.

> ⚠ **Attention**
>
> n punti di decisione in sequenza possono dar luogo (non necessariamente) a 2^n possibili cammini distinti

> ⚠ **Attention**
>
> un ciclo iterativo while o chiamate ricorsive possono dar luogo a infiniti cammini distinti, si pongono dei limiti di copertura dei possibili path.
> Ad esempio nel caso di cicli while i tipici path testati sono almeno tre: 0 cicli, 1 ciclo iterativo, 2 (o più) cicli iterativi.

Data Flow Coverage

Il criterio di **data flow coverage** comprende la selezione di un test set che *copre il più possibile i cammini def-use delle variabili*

- Def: espressione in cui la variabile viene assegnata
- Use: espressione in cui la variabile viene usata