Software e difetti

- Il software con difetti è un grande problema
- I difetti nel software sono comuni

- Come sappiamo che il software ha qualche difetto?
  - Conosciamo tramite 'qualcosa', che non è il codice, cosa un programma dovrebbe fare
  - Tale 'qualcosa' è una specifica
  - Tramite il comportamento anomalo, il software sta comunicando qualcosa -> i suoi difetti -> questi non devono passare inosservati

Verifica e Validazione (V & V)

- Obiettivo di V & V: assicurare che il sistema software soddisfi i bisogni dei suoi utenti
- Verifica
  - Stiamo costruendo il prodotto nel modo giusto?
  - Il sistema software dovrebbe essere conforme alle sue specifiche
- Validazione (convalida)
  - Stiamo costruendo il giusto prodotto?
  - Il sistema software dovrebbe fare ciò che l'utente ha realmente richiesto

Processo di V & V

- Si dovrebbe applicare il processo di V&V ad ogni fase durante lo sviluppo
- Il processo di V&V ha due obiettivi principali: scoprire i difetti del sistema e valutare se il sistema è usabile in una situazione operativa
- I difetti possono essere raggruppati, in base alle fasi di sviluppo
  - Difetti di specifiche: la descrizione di ciò che il prodotto fa è ambigua, contraddittoria o imprecisa
  - Difetti di progettazione: le componenti o le loro interazioni sono progettati in modo non corretto, le cause: algoritmi (es. divisione per zero), strutture dati (es. campo mancante, tipo sbagliato), interfaccia moduli (parametri di tipo inconsistente), etc.
  - Difetti di codice: errori derivanti dall'implementazione dovuti a poca comprensione del progetto o dei costrutti del linguaggio di (es. overflow, conversione tipo, priorità delle operazioni aritmetiche, variabili non inizializzate, non usate tra due assegnazioni, etc.)
  - A volte è difficile classificare se un difetto è di progettazione o di codice
  - Difetti di test: i casi di test, i piani per i test, etc. possono avere difetti

Test

- Il test del software
  - Può rivelare la presenza di errori, non la loro assenza
  - Un test ha successo se scopre uno o più errori
  - I test dovrebbero essere condotti insieme alle verifiche sul codice statico
  - La fase di test ha come obiettivo rivelare l'esistenza di difetti in un programma
- Il debugging si riferisce alla localizzazione ed alla correzione degli errori
- Debugging
  - Formulare ipotesi sul comportamento del programma
  - Verificare tali ipotesi e trovare gli errori

Test: definizioni

- Dati di test (test data)
  - Dati di input che sono stati scelti per testare il sistema
- Casi di test (test case)
  - Dati di input per il sistema e output stimati per tali input nel caso in cui il sistema operi secondo le sue specifiche
    - Gli input sono non solo parametri da inviare ad una funzione, ma anche eventuali file, eccezioni, e stato del sistema, ovvero le condizioni di esecuzione richieste per poter eseguire il test
- Test suite: insiemi di casi di test

Testing

- Solo un test esaustivo può mostrare se un programma è privo di difetti
  - I test esaustivi sono impraticabili
    - Es. Una funzione che prende in ingresso 2 int, per essere testata esaustivamente dovrebbe essere eseguita 2^32*2^32 volte, ovvero circa 1.8 * 10^19 volte
    - Se la funzione esegue in 1ns = 10^-9s occorrono 1.8*10^10s ovvero, essendo 1Y = 3 * 10^7, occorrono circa 600 anni!
- Priorità
  - I test dovrebbero mostrare le capacità del software più che eseguire i singoli componenti
  - Il test delle vecchie funzionalità è più importante del test delle nuove
  - Testare situazioni tipiche è più importante rispetto a testare situazioni limite

Princìpi per i Test

- Fare test è il processo che esercita il componente usando un set selezionato di casi di test con l'intento di (i) rivelare difetti, (ii) valutare la qualità
- Quando l'obiettivo è trovare difetti allora un buon test case è uno che ha buona probabilità di rivelare difetti non noti
- Un caso di test deve contenere i risultati aspettati (output stimati)
- I casi di test dovrebbero essere sviluppati sia per condizioni di input valide che non valide
- La probabilità di esistenza di difetti addizionali per un componente software è proporzionale al numero di difetti già individuati per il componente. I difetti spesso accadono in gruppi. Un codice che ha grande complessità ha una progettazione non buona
- I test devono essere ripetibili e riusabili: test di regressione

Strategie di Test

- Un approccio in cui i test vengono effettuati senza conoscere come è fatto il sistema (né la struttura, né il codice) si dice test black-box, ovvero considera il sistema una scatola nera
  - I casi di test sono progettati sulla base della descrizione del sistema, ovvero partendo dal documento di specifiche del sistema
    - E' possibile studiare (e predisporre) i test nelle fasi iniziali dello sviluppo del software
  - Dall'insieme dei dati di input possibili si individua il sottoinsieme che può rivelare la presenza di difetti nel sistema in modo da progettare casi di test efficaci
- Un altro approccio è quello white-box (detto anche glass-box o strutturale) che focalizza sulla struttura del software da testare, bisogna avere a disposizione il codice sorgente (o una opportuna rappresentazione tramite pseudo-codice)
- Entrambi gli approcci sono usati per rendere la fase di test più efficace

Partizionamento in classi equivalenti

- Nel caso di test black-box, un buon modo per selezionare gli input per il test è ricorrere a partizioni in classi equivalenti
- Dati di input e risultati si possono spesso raggruppare in classi (categorie) in cui tutti i membri di una classe sono relazionati. Ognuna delle classi è una partizione equivalente, ovvero mi aspetto che il programma effettui elaborazioni simili (equivalenti) per ciascun membro della stessa classe
- Testare uno dei valori membri di una classe equivale a testare ciascun altro valore della stessa classe: viene meno la necessità di test esaustivi
  - Permette di coprire un grande dominio con un piccolo set di valori
- I casi di test dovrebbero essere scelti da ciascuna partizione. Es. Una funzione può prendere in input solo numeri da 4 a 20
  - Partizioni: numeri <4; numeri tra 4 e 20; numeri >20
  - Dati di test da scegliere: 3, 4, 12, 20, 21 (alcuni numeri sono scelti per i test di confine: boundary)
- Chi fa il test deve considerare sia classi di equivalenza valide che classi di equivalenza non valide. Una classe di equivalenza non valida rappresenta input inaspettati o errati

Test del percorso

- La finalità è assicurare che i casi di test siano tali che ogni percorso all'interno del programma sia eseguito almeno una volta
- E' utile rappresentare il programma tramite un grafo di flusso dove i nodi rappresentano condizioni del programma e gli archi il flusso di controllo
- Complessità ciclomatica (cc) = numero di archi - numero di nodi + 2
  - Per testare tutti le condizioni, il numero di test da effettuare è cc
  - Tutti i percorsi sono eseguiti, ma non tutte le combinazioni dei percorsi

Test sotto stress

- Eseguire il sistema oltre il massimo carico previsto consente di rendere evidenti i difetti presenti
- Il sistema eseguito oltre i limiti consentiti non dovrebbe fallire in modo catastrofico
- Test di stress indagano su perdite, di servizio o dati, ritenute inaccettabili
- Particolarmente rilevanti per i sistemi distribuiti che possono subire degradazioni in dipendenza delle condizioni della rete
- PS: completare le specifiche in accordo ai risultati dei test

Copertura del codice

- Copertura (coverage): misura (in percentuale) della parte del codice sorgente di un programma che è eseguita quando una test suite è eseguita
- Fino a quando dovremmo continuare a fare test?
- Metrica: Copertura del codice (Code coverage)
  - Dividere il programma in unità (es. costrutti, condizioni, comandi)
  - Definire la copertura che dovrebbe avere la suite di test (es. 60%)
  - Copertura codice = numero di unità già eseguite / numero di unità del programma
- Si smette di eseguire test quando si è raggiunta la copertura desiderata
- Avere una copertura del 100% non significa non avere difetti
  - Pensare ad esempio ai dati di input scelti
- Parti critiche del sistema possono avere copertura maggiore di altre parti
- La misura di copertura permette di capire se alla suite di test manca qualcosa

Criteri di Copertura del Codice

- **Copertura di funzioni**: indica la percentuale di funzioni eseguite
- **Copertura di costrutti (statement)**: indica la percentuale di costrutti eseguiti
- **Copertura di archi**: indica la percentuale di archi del grafo di controllo del flusso (control-flow graph) eseguiti
  - **Copertura di rami**: indica la percentuale di rami condizionali eseguiti (per es. sia il blocco if che else)
- **Copertura di condizioni**: valuta se ogni sotto-espressione boolean è stata valutata sia true che false.
- Es. if (x and y) then a = 1 else a = 0
- Una combinazione di copertura di funzioni e di rami è chiamata **copertura di decisioni**. Si ottiene quando ciascun punto di ingresso e di uscita nel programma è stato eseguito almeno una volta e quando ogni espressione boolean è stata valutata con tutti i possibili risultati

Trend di difetti scoperti

- Metrica bug trend: misura la frequenza con cui i difetti sono trovati
  - Quando la frequenza tende a zero
    - Non ci sono più difetti
    - Drammatico aumento dei costi di ricerca dei difetti
- Pratiche standard
  - Eseguire i test spesso e lavorare con nuove versioni (nightly build)
  - Fare progressi in avanti (regression test: eseguire i test già scritti per prevenire l'apparizione di difetti già eliminati)
  - Condizioni di stop (coverage, bug trend)