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

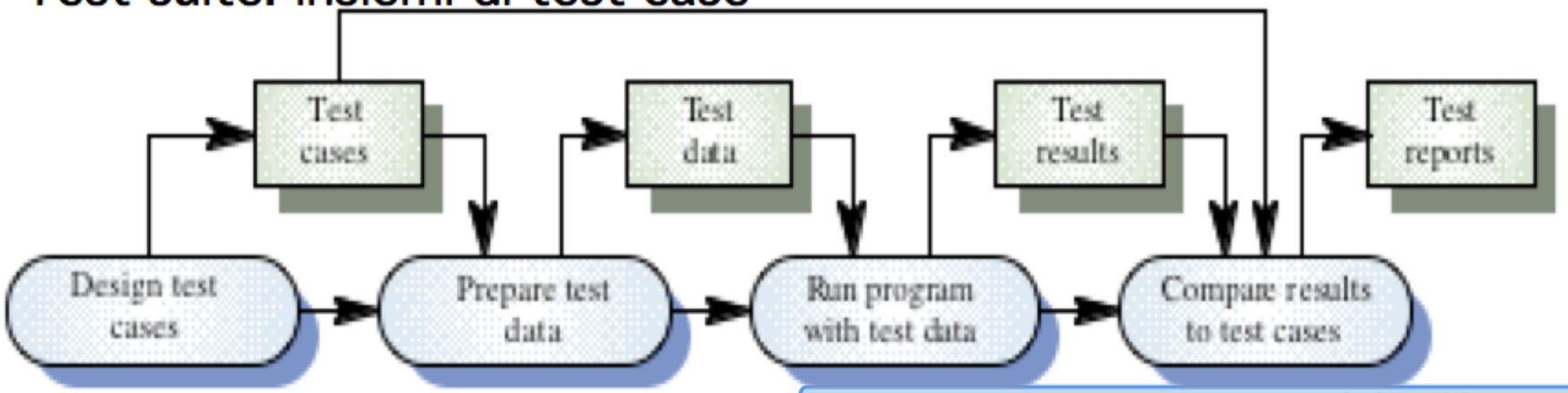
- Si dovrebbe applicare il processo di V&V ad ogni fase durante lo sviluppo
- Il processo di V&V ha due obiettivi principali: scoprire i difetti del sistema e valutare se il sistema è usabile in una situazione operativa
- I difetti possono essere raggruppati, in base alle fasi di sviluppo
  - Difetti 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 test case



Difficoltà per chi fa i test (tester)

- Deve avere una conoscenza vasta delle discipline di ingegneria del software
- Deve avere conoscenza ed esperienza su come un software è descritto (specifiche), progettato e sviluppato
- Deve essere in grado di gestire molti dettagli
- Deve conoscere quali tipi di fault possono generare i costrutti del codice
- Deve ragionare come uno scienziato per proporre ipotesi che spiegano la presenza di tipi di difetti
- Deve avere una buona comprensione del dominio del software
- Deve creare e documentare casi di test, quindi selezionare gli input che con maggiore probabilità possono rivelare difetti
- Necessita di lavorare e cooperare con chi si occupa di requisiti, design, sviluppo codice e spesso con clienti ed utenti

Testing

- L'obiettivo del testing è di stabilire la presenza di difetti nei sistemi
  - Un test ha successo se il test fa sì che il programma si comporti in modo anomalo
- Test dei componenti (detti anche unit test)
  - Test dei singoli frammenti (metodi, classi, etc.)
  - Questo tipo di test è effettuato dallo sviluppatore del componente
  - Come progettare i test? In base a tecniche note ed all'esperienza dello sviluppatore
- Test di integrazione
  - Test di gruppi di componenti già integrati (interagenti) che formano un sistema o un sottosistema
  - La responsabilità è di un team di test
  - I test sono basati sulle specifiche

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, 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

Strategie di Test

- Un approccio in cui i test vengono effettuati senza avere conoscenza di come il sistema è fatto (ovvero della sua struttura interna) 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 che focalizza sulla struttura interna 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 efficiente

Partizionamento in classi equivalenti

- Nel caso di test black-box, un buon modo per selezionare gli input per il test al sistema 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

Partizionamento

- 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
- Classi di equivalenza possono essere selezionate anche per le condizioni di output
- Non ci sono regole forti per individuare le classi di equivalenza => il partizionamento è un processo euristico, tester diversi potrebbero individuare classi diverse
- Può essere difficile identificare classi di equivalenza

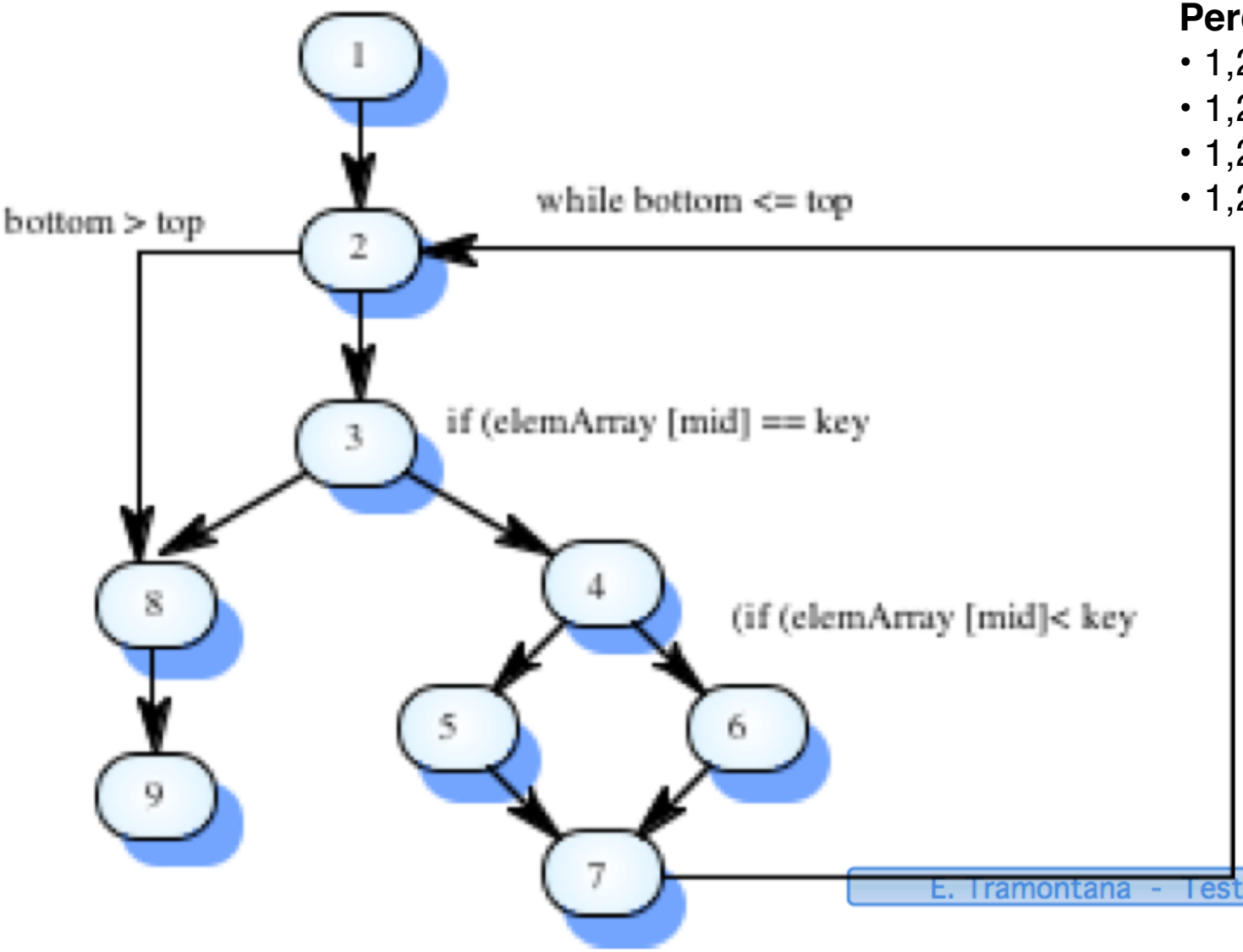
Test strutturali

- Chiamati anche test white-box, glass-box, o clear-box
- Test (addizionali a quelli black-box) derivati dalla conoscenza della struttura del programma
- La finalità è di eseguire tutti i costrutti del programma (non tutte le combinazioni dei percorsi)

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

Percorsi indipendenti



Percorsi indipendenti:

- 1,2,8,9
- 1,2,3,8,9
- 1,2,3,4,5,7,2,8,9
- 1,2,3,4,6,7,2,8,9

Un percorso si dice *indipendente* se attraversa almeno un arco non ancora percorso prima di definire il percorso in questione

es. 1,2,3,4,5,7,2,3,8,9 non è un percorso indipendente, essendo una *combinazione* di percorsi già specificati

Percorsi indipendenti

- I casi di test dovrebbero essere scelti in modo che tutti i percorsi indipendenti siano eseguiti
- Un tool può essere usato a runtime per controllare che i percorsi siano stati eseguiti

Test di integrazione

- Sono test eseguiti su sistemi completi o su sottosistemi
- I test di integrazione dovrebbero essere black-box e derivati dalle specifiche
- La principale difficoltà è di localizzare gli errori
  - Effettuare i test di integrazione in maniera incrementale riduce tale problema
- Per i test di integrazione incrementali
  - Sull'insieme dei componenti A, B si eseguono le suite di test T1, T2, T3, successivamente
  - Sull'insieme di componenti A, B, C si eseguono le suite di test T1, T2, T3, T4, etc.

Approcci per i test di integrazione

- Top down
  - Integrare i sotto-sistemi (componenti) di più alto livello e successivamente quelli dei livelli un pò più bassi
    - Sostituire i componenti con stub quando appropriato
  - Permette di scoprire errori nell'architettura del sistema
  - Permette di mettere a punto versioni demo nelle fasi iniziali
- Bottom-up
  - Integrare singoli componenti di basso livello e successivamente tali integrazioni con componenti di livello più alto
  - Rende la scrittura dei test più semplice
- In pratica, ciò che avviene è una combinazione dei due precedenti approcci

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

Test sotto stress

- Stress
  - Prestazioni: inserire i dati con frequenza molto alta, o molto bassa
  - Strutture dati: funziona per qualsiasi dimensione dell'array?
  - Risorse: test con poca memoria RAM, numero basso di file che possono essere aperti, connessioni di rete, etc.

Testing manuale

- I casi di test sono liste di istruzioni per una persona
  - Click su “login”
  - Inserisci username e password
  - Click su “ok”
  - Inserisci il dato ...
- Molto comune, poiché
  - Non sostituibile: test di usabilità
  - Non pratico da automatizzare: troppo costoso
  - Le persone che fanno i test non sanno gestire automatismi complessi

Testing automatico

- Registrare un test manuale e rieseguirlo automaticamente
  - Con macro, script, programmi appositi (es. AutoHotkey)
  - Spesso poco robusto
    - Smette di funzionare se cambia qualcosa dell'ambiente (es. posizione campi, nome campi, etc.)
- Sviluppare programmi che eseguono il test sul codice
  - Chiamano funzioni, confrontano risultati, etc.

Test regressivi

- Linee guida
  - Scoperto un difetto
  - Costruire un test che permette di rilevare il difetto
  - Eseguire lo stesso test tutte le volte che il codice viene cambiato
  - Il difetto non riappare
- I test regressivi assicurano di non ritornare a versioni che presentano difetti già corretti
- In pratica, eseguo spesso i test già scritti, se la loro esecuzione non ha durata proibitiva
  - Ciascun test dovrebbe durare il meno possibile

Copertura del codice

- 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