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
  - *Stiamo costruendo il giusto prodotto?*
  - Il sistema software dovrebbe fare ciò che l'utente ha realmente richiesto

Processo di V & V

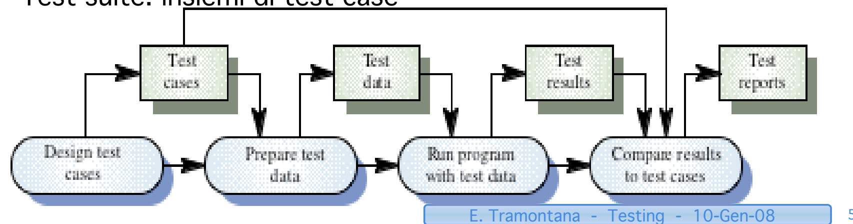
- Il processo di verifica e validazione dovrebbe essere applicato ad ogni fase durante lo sviluppo
- Il processo di V & V ha due obiettivi principali: scoprire i difetti del sistema e valutare se il sistema è usabile in una situazione operativa
- I difetti possono essere raggruppati in classi, in accordo alle fasi di sviluppo del software
  - Difetti di specifiche: la descrizione di ciò che il prodotto fa è ambigua, contraddittoria o imprecisa
  - Difetti di design: le componenti o le interazioni tra queste sono progettati in modo non corretto per la progettazioni di 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 design o dei costrutti del linguaggio di programmazione (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 design 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



Princìpi per i test

- Fare test è il processo di esercitare 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
- I risultati dei test dovrebbero essere letti meticolosamente
- Un caso di test deve contenere i risultati aspettati
- 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 del componente già individuati
  - I difetti spesso accadono in gruppi
  - Un codice che ha grande complessità ha un cattivo design

Princìpi per i test

- I test dovrebbero essere effettuati da un gruppo indipendente dal gruppo di sviluppatori
  - Gli sviluppatori sono orgogliosi del codice prodotto, inoltre possono avere difficoltà a capire dove trovare difetti, poiché il loro modello mentale oscura il codice reale
- I test devono essere ripetibili e riusabili
  - Test di regressione
- I test dovrebbero essere pianificati
  - I piani di test dovrebbero specificare gli obiettivi, allocare tempo e risorse umane, monitorare i risultati
- Le attività di test dovrebbero essere integrate nel ciclo di sviluppo del software

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 un 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

Partizionamento: Lista di condizioni

1. Se una condizione per l'input è specificata come un range di valori ammessi (o un numero di valori contigui), selezionare una classe valida costituita dal range e due classi invalide, ciascuna ad un estremo del range
2. Se una condizione per l'input è data da un numero di valori, selezionare una classe valida che include i valori consentiti e due invalide per i valori fuori da ciascun estremo del set
3. Se una condizione per l'input è data da un set di valori, selezionare una classe valida costituita dai valori e una invalida per i valori fuori dal set
4. Se una condizione per l'input è descritta come "deve essere" considerare due classi, una valida che rappresenta la condizione "deve essere" e una invalida che non include la condizione "deve essere"
   - Es. il testo deve iniziare con una vocale
5. Se si crede che un elemento di una classe di equivalenza non verrà trattato in modo identico agli altri elementi della classe allora la classe deve essere ulteriormente partizionata in classi di equivalenza più piccole

Partizionamento Es. Funzione `search()`

- Specifiche funzione `search()`
  - Input
    - k di tipo `Elem` da cercare, c array di `Elem` su cui effettuare la ricerca
  - Output
    - `t` boolean e `i` indice dell'array c; se k è trovato, `t` è `true` e `i` indica la posizione nell'array c
  - Precondizioni
    - c ha almeno un elemento; c è ordinato in maniera crescente, ovvero `c[j]<=c[n]` per `j,n>0` e `j<=n`
  - Postcondizioni
    - Se k è in c, allora `t=true` e `c[i]=k`

    oppure
    - Se k non è in c, allora `t=false` e non esiste `i` tale che `c[i]=k`

Partizionamento Es. Funzione `search()`

- Partizionamento degli input in classi equivalenti
  - Input che sono conformi alle precondizioni (classi valide)
    - c contiente 1 elemento
    - numero di elementi di c > 1, inoltre c è ordinato in maniera crescente
  - Input per cui una precondizione non è soddisfatta (classe invalida)
    - c non contiene nessun elemento
  - Input dove k è un elemento dell'array (classi valide)
    - k è il primo elemento (dovuto a boundary value analysis)
    - k è l'ultimo elemento (dovuto a boundary value analysis)
    - k è un altro elemento, diverso dal primo o dall'ultimo
  - Input dove k non è un elemento dell'array (classe valida)
    - k non è presente in c

Partizionamento Es. Funzione search()

- Dopo aver identificato tutte le classi di equivalenza
  - Assegnare un identificatore a ciascuna classe di equivalenza
  - Sviluppare dei casi di test per coprire ciascuna classe di equivalenza, un caso di test potrebbe coprire più di una classe di equivalenza

Partizioni per la funzione search()

- Le singole condizioni per test individuati sono combinate tra loro

Classe	***Test***	***Array c***	***valore k***
valida	T1	singolo valore	k non presente
valida	T2	singolo valore	k presente
valida	T3	più valori	k non presente
valida	T4	più valori	k è il primo elemento di c
valida	T5	più valori	k è l'ultimo elemento di c
valida	T6	più valori	k è l'elemento di mezzo di c
invalida	T7	nessun valore	K non presente

Test	***Array c***	***valore k***	***output*** (t, i)
T1	5	3	false, ??
T2	5	5	true, 1
T3	2, 3, 6, 9, 10	1	false, ??
T4	2, 3, 6, 9, 10	2	true, 1
T5	2, 3, 6, 9, 10	10	true, 5
T6	2, 3, 6, 9, 10	6	true, 3

Esempio ID

- Specifica: un identificatore deve avere da 3 a 15 caratteri alfanumerici ed i primi due caratteri devono essere lettere
  - Notare il “deve essere”
- Partizioni
  - Nome alfanumerico, classe valida
  - Nome non alfanumerico, classe invalida
  - Nome tra 3 e 15 caratteri, classe valida
  - Nome con meno di 3 caratteri, classe invalida
  - Nome con più di 15 caratteri, classe invalida
  - Primi due caratteri lettere, classe valida
  - Primi due caratteri non lettere, classe invalida

Linee guida per i test di sequenze

- Scegliere sequenze che hanno un solo valore
- Usare sequenze di dimensioni diverse nei vari test
- Far sì che il primo, il medio e l’ultimo elemento della sequenza siano acceduti
- Testare con sequenze di lunghezza zero

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)

search()

```
public class BinSearch {
  public void search(int key, int[] elemArray, Result r) {
    int bottom = 0;
    int top = elemArray.length -1;
    int mid;
    r.found = false;
    r.index = -1;
    while (bottom <= top) {
      mid = (top + bottom)/2;
      if (elemArray[mid] == key) {
        r.found = true;
        r.index = mid;
        return;
      } else if (elemArray[mid] < key) bottom = mid+1;
             else top = mid-1;
    }
  }
}
```


Partizioni equivalenti

- Dalla conoscenza dell'algoritmo
  - Partizioni equivalenti
    - Precondizioni soddisfatte, k presente
    - Precondizioni soddisfatte, k non presente
    - Precondizioni non soddisfatte, k presente
    - Precondizioni non soddisfatte, k non presente
    - L'array ha dimensione 1
    - L'array ha un numero pari di valori
    - L'array ha un numero dispari di valori
  - Partizioni equivalenti per l'array
    - K è presente sulla parte dal primo elemento all'elemento medio-1
    - K è l'elemento medio
    - K è presente sulla parte dall'elemento medio+1 all'ultimo elemento

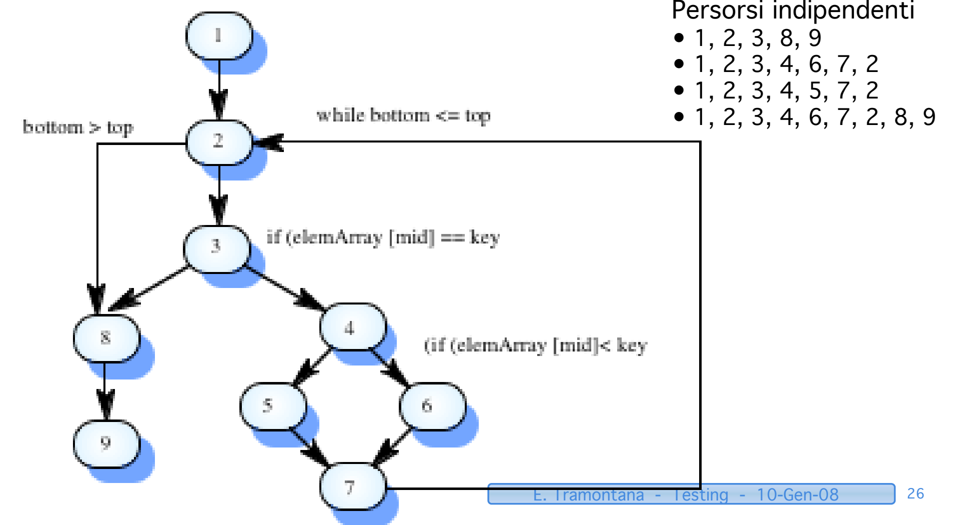
Casi di test

Array c	***valore k***	***output*** (t, i)
5	3	false, ??
5	5	true, 1
2, 3, 6, 9, 10	1	false, ??
2, 3, 6, 9, 10	2	true, 1
2, 3, 6, 9, 10	10	true, 5
2, 3, 6, 9, 10	6	true, 3
2, 3, 6, 9, 10	3	true, 2
2, 3, 6, 9, 10, 15	10	true, 5

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



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

Percorsi indipendenti

- I casi di test dovrebbero essere scelti in modo che tutti i percorsi 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

Trend di difetti scoperti

- 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)
  - Condizioni di stop (coverage, bug trend)

Test dei sistemi OO

- Quando un sistema OO si può testare?
- Il testing è definito in termini di osservabilità e controllabilità
  - Per testare un componente, è necessario poter controllare gli input (e lo stato) e osservare i suoi output
  - Se non è possibile controllare gli input, non è possibile determinare con certezza cosa ha causato l'output
  - Se non è possibile osservare l'output non si ha la certezza di come è stato elaborato un certo input
- Information hiding e ereditarietà possono ridurre osservabilità e controllabilità
  - Design by contract può essere una soluzione
  - Le precondizioni permettono di osservare e controllare gli input
  - Le postcondizioni permettono di osservare gli output

Test dei sistemi OO

- I componenti da testare sono classi e oggetti
  - Hanno granularità più grande delle funzioni
  - Non è sempre evidente quali classi sono di più alto livello, quindi difficoltà ad effettuare testing top-down
- Test
  - Prima i metodi
    - Analogo per lo più al test di funzioni (già visto)
  - Poi le classi (superclassi)
    - Test di sequenze di invocazioni di metodo
    - Per evidenziare attributi usati e stati dell'oggetto
  - Poi le sottoclassi
    - Test dei metodi aggiunti, e di quelli ridefiniti (override)
    - Se ci sono metodi override o metodi che cambiano lo stato della superclasse il test coinvolge la superclasse
  - Poi gruppi di classi
    - Far riferimento ai casi d'uso

Classi e stato

- Le classi sono caratterizzate da uno stato
- I test devono essere eseguiti per i diversi stati della classe
  - Prima, portare la classe nello stato opportuno
  - Poi, invocare i metodi
- La specifica può dare informazioni sullo stato
- Esempio
  - Stati di una classe: NotInstalled, Bound, Unbound
  - Transizioni considerate possibili (lecite)
    - Install: da NotInstalled a Unbound
    - Bind: da Unbound a Bound
    - Unbind: da Bind a Unbound
- Per fare i test, tramite invocazioni di metodi, attraversare:
  - Ogni transizione
  - Tutti i percorsi (leciti) tra gli stati

Test isolati

- L'esecuzione di ciascun test dovrebbe essere isolata da esecuzioni di altri test
  - Rende possibile l'esecuzione di un test indipendentemente dagli altri test già eseguiti
  - Lascia il sistema in uno stato consistente, così da evitare fallimenti in cascata quando si esegue una sequenza di test
    - Per ciascun test si dovrebbe: inizializzare lo stato del sistema, eseguire il test, ripristinare lo stato del sistema
  - Test isolati sono indipendenti dall'ordine di esecuzione
  - Durante la produzione di test isolati si scopre se il sistema da testare è stato realizzato in maniera modulare
    - Costruire test isolati, in fasi iniziali dello sviluppo, aiuta a strutturare in modo modulare il sistema da realizzare

Implementazione Test

- Ricavare una lista dei test che si vuole effettuare
- Implementare i test, uno alla volta
  - Eseguire il test, prima di passare al successivo test da implementare
  - Fare refactoring per correggere eventuali difetti scoperti dall'esecuzione
  - Implementare i successivi test per il codice che si ottiene dopo il refactoring

Testing Pattern

- Child Test
  - Se un test è troppo grande (tanti settaggi da fare per il sistema, tante funzioni da invocare, molto codice da scrivere)
  - Generare più test a partire da questo, cercando di capire perché è troppo grande
- Mock Object
  - Come testare un codice che dipende da risorse costose o complicate?
  - Creare un sostituto (mock) della risorsa che restituisce costanti
  - Es. di risorsa: un database. Complicato poiché: dovrà essere riempito opportunamente, ha tempi di risposta lunghi, è in remoto

```
Database db = new MockDB();
db.query("select …");
```


Testing Patterns

- Log String
  - Come assicurarsi che una sequenza di chiamate è avvenuta secondo l'ordine corretto?
  - Inserire il log in una stringa ed appendere mano a mano alla stringa

```
public void setUp(Logger log) {
  log.s = "setUp ";
  result = testMethod();
  log.s = log.s + "testMethod ";
  …
}
```


Testing Patterns

- Crash Test Dummy
  - Come verificare se il sistema si comporta bene in situazioni di fault poco probabili (es. file system pieno)?
  - Simulare la situazione di fault

```
public class FullFile extends File {
  public FullFile(String path) {
    super(path);
  }
  public boolean createNewFile() throws IOException {
    throw new IOException();
  }
}

File f = new FullFile("foo");
try {
  createNewFile();
} catch (IOException e) { ... }
```