17. Testing del software

Paola Barra
2022/2023

Cosa studieremo del testing

- Terminologia
  - Guasto, errore, difetto
- Attività di Test
  - **Testing delle unità**
  - Testing di integrazione
  - Testing del sistema

I bug...

- Nella storia dell'informatica ci sono innumerevoli casi di sistemi con comportamenti che deviano dalle specifiche
  - Primo caso di bug (e debug) della storia:
    - Nel 1945, una falena (bug) entrò nel Pannello F, Relay #70 del computer Harvard Mark II, creando un cortocircuito
      - Il computer iniziò a sbagliare addizioni e moltiplicazioni
    - La falena (ben cotta?) fu presa (de-bug) ed attaccata sul logbook del computer con dello scotch, con la scritta "primo caso di bug trovato" ("first actual case of a bug being found")

“Errori” dal passato…

- **Errori di progettazione**
- **NASA Mars Climate Orbiter**
  - La sonda spaziale Mars Climate Orbiter, costata 125 milioni di dollari, fu persa il 23 settembre 1999, mentre entrava nell’orbita di Marte
  - Si scoprì successivamente che la causa era in un errore insito nelle specifiche del software adottate dalla NASA
  - Durante il processo di sviluppo gli analisti non avevano specificato il sistema di riferimento da usare per la gestione dei motori usati nell’atterraggio!!!
  - Conseguenza: un team di sviluppo utilizzò il sistema Imperiale (pounds), mentre un altro, il sistema Metrico (Kg)
    - Quando furono passati i parametri da un modulo sw all’altro, non fu operata conversione dei dati, con il risultato di dare direzioni di volo completamente errate, che portarono allo schianto della sonda

“Errori” dal passato…

- **Errori di coding**
- **ESA Ariane 5**
  - Il vettore spaziale Ariane 5 esplose durante il suo volo inaugurale, il 4 Giugno 1996
  - Per la gestione dei motori, l’Ariane 5 utilizzava il codice funzionante correttamente sul suo predecessore Ariane 4
    - Utilizzava una routine aritmetica di conversione da un floating point a 64 bit ad un intero a 16 bit
  - I valori generati dai motori dell’Ariane 4 erano gestibili da interi a 16 bit
  - I motori dell’Ariane 5, più potenti, generavano valori più grandi
  - Il conseguente overflow durante la conversione portò al crash sia del sistema primario che secondario di navigazione, con la conseguente esplosione del razzo

“Errori” dal passato…

- **Errori nella gestione dell’input dell’utente**
- **USS Yorktown CG-48**
  - Nel 1998, durante delle esercitazioni in mare aperto, un sottoufficiale dell’incrociatore lanciamissili USS Yorktown inserì per sbaglio uno zero in un form del Data Base Manager di bordo
    - Ciò portò ad una divisione per zero
  - L’errore bloccò tutti i terminali della LAN della nave, arrivando allo shutdown di tutti i sistemi di bordo, comprese le turbine
  - La nave restò completamente immobilizzata per ore, in pieno oceano, perché un form per l’inserimento di dati in un DB non verificava la correttezza dell’input inserito dall’utente

“Errori” dal passato

- **Errori di valutazione sull’utilizzo del sistema**
- **NASA Spirit Rover**
  - Il *robottino* per l’esplorazione di Marte “Spirit” improvvisamente cessò di comunicare con la Terra il 21 Gennaio 2004, qualche giorno dopo l’atterraggio
  - Nei giorni successivi, i tecnici della NASA riuscirono a scoprire che il robot era in Fault mode, e si riavviava in continuazione
  - Poiché l’errore si ripresentava ad ogni reboot, i tecnici capirono che era un problema della Flash Memory o una rottura hardware
  - Isolando la Flash attraverso un comando speciale, il robot si riavviò senza problemi
  - Dopo 3 giorni, il team capì l’errore: il robot aveva tenuto traccia dei principali parametri sia durante il volo fino a Marte, che durante i test al suolo
    - Come risultato, aveva creato un enorme numero di file, che portava al crash del modulo che si occupava di gestire il File System
  - Dopo aver riformattato la Flash Memory, il robottino ha funzionato correttamente per anni

Incidenti famosi

- **F-16**: attraversamento dell'equatore usando il pilota automatico
  - Risultato: aereo ribaltato
  - Motivo: riuso del pilota automatico di un razzo

- Gli incidenti di **Therac-25** (1985-1987), probabilmente in assoluto il più grave fallimento non militare legato al computer in termini di vita umana (almeno cinque morti)
  - Motivo: cattiva gestione evento nella GUI

L’incidente del Therac-25

- Nel 1986, un paziente andò in clinica per ricevere il suo solito trattamento a bassa radiazione alla sua spalla
- Il tecnico digitò "X" (fascio raggi x), accortosi dell‘errore, velocemente modificò in "E" (fascio elettroni) e premette “enter" :
  - X <Delete char> E <enter>
  - Tale sequenza di input non fu mai testata in un lasso temporale breve (circa 8 secondi)
- Therac-25 segnalò "fascio pronto" mostrando all‘operatore la modalità a bassa energia
- Il tecnico digitò B per avviare il fascio di energia
  - Il fascio che invece uscì dal macchinario fu un’esplosione di 25000 rad con 25 milioni di elettronvolt, 125 volte maggiore di una dose regolare
  - La macchina rispose con un messaggio di errore "Malfunction 54", non spiegato nel manuale utente
    - La macchina ha mostrato un sotto dosaggio
  - L'operatore premette "P" per continuare per ulteriori trattamenti. Ancora una volta, lo stesso messaggio di errore
- Il paziente sentiva forti dolori alla schiena, molto diversi dal suo solito trattamento. È morto 3 mesi dopo

Motivi del guasto di Therac-25

- Fallimento nel riuso corretto del vecchio software di Therac-6 e Therac-20 in Therac-25
- Messaggi di warning criptici
- Gli utenti non capivano il problema ricorrente (5 pazienti deceduti)
- Mancanza di comunicazione tra l'ospedale ed i costruttori
- Il costruttore non prendeva in considerazione che la macchina potesse sbagliare
- Nessun hardware adeguato per rilevare problemi di sicurezza

Come fu risolto il problema

- Il 10 febbraio 1987, lo Health Protection Branch del governo Canadese e il Food and Drug Administration degli USA, annunciarono che Therac-25 era pericoloso da usare
- Il 21 luglio del 1987 furono date istruzioni dalla AECL* su come riparare Therac-25. Alcuni delle istruzioni furono
  - Gli operatori non possono riavviare la macchina senza reinserire il comando di input
  - La dose da somministrare al paziente deve essere mostrata chiaramente all'operatore
  - Limitare le modalità di input per prevenire errori accidentali
  - I messaggi di errore devono essere resi più chiari
  - Tutti i manuali devono essere riscritti per includere le nuove modifiche

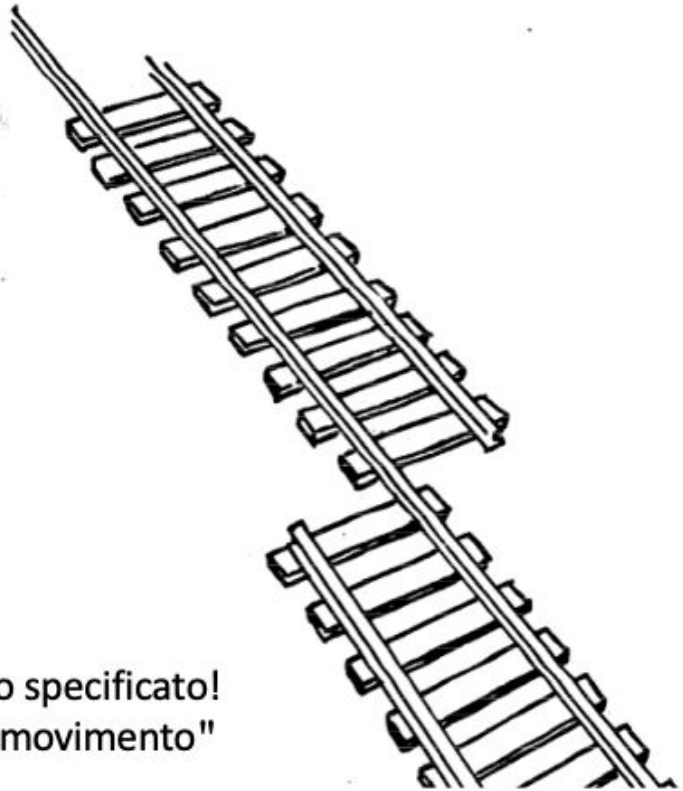
*Atomic Energy of Canada Limited

Terminologia

- Guasto (failure): qualsiasi differenza del comportamento osservato rispetto al comportamento specificato
- Stato d'errore (errore): il sistema è in uno stato tale che ulteriori elaborazioni fatte dal sistema portano ad un guasto
- Difetto (fault o bug): la causa meccanica o algoritmica di un errore
- Validazione: attività di controllo delle differenze tra comportamento osservato di un sistema e la sua specifica

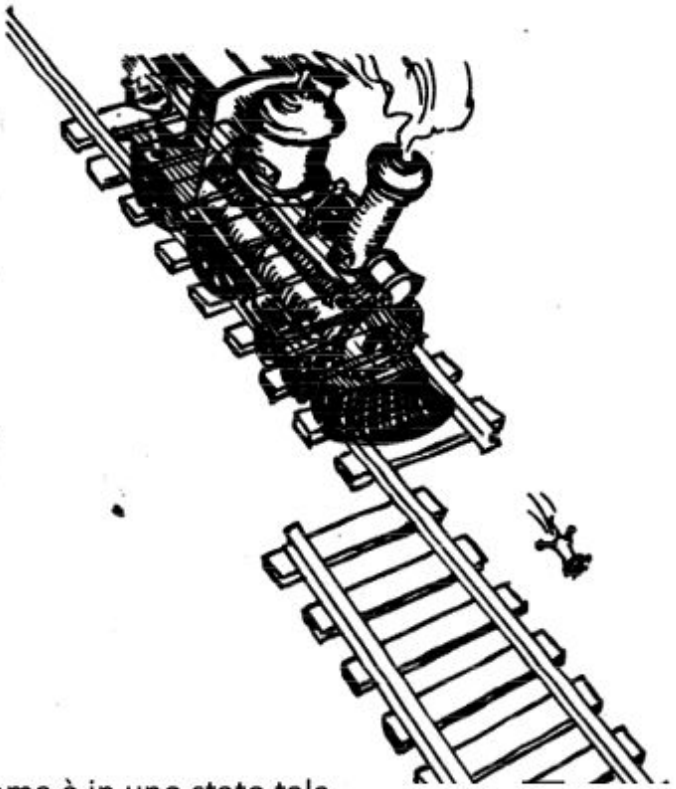
Come lo definireste?

- Un guasto?
- Un errore?
- Un difetto?



Dobbiamo prima descrivere il comportamento specificato!
Specifica: "un binario supporterà un treno in movimento"

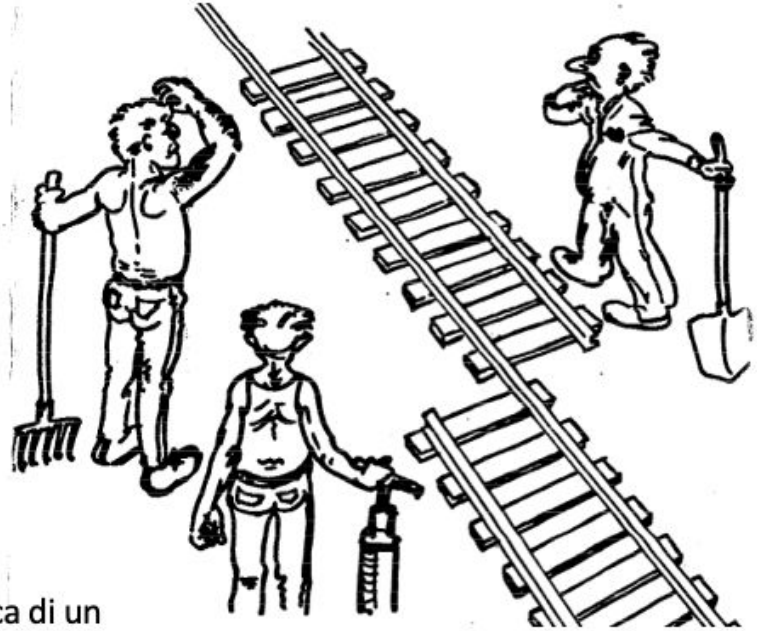
Stato d’errore (Errore)



Stato d’errore (errore): il sistema è in uno stato tale che ulteriori elaborazioni fatte dal sistema portano ad un guasto

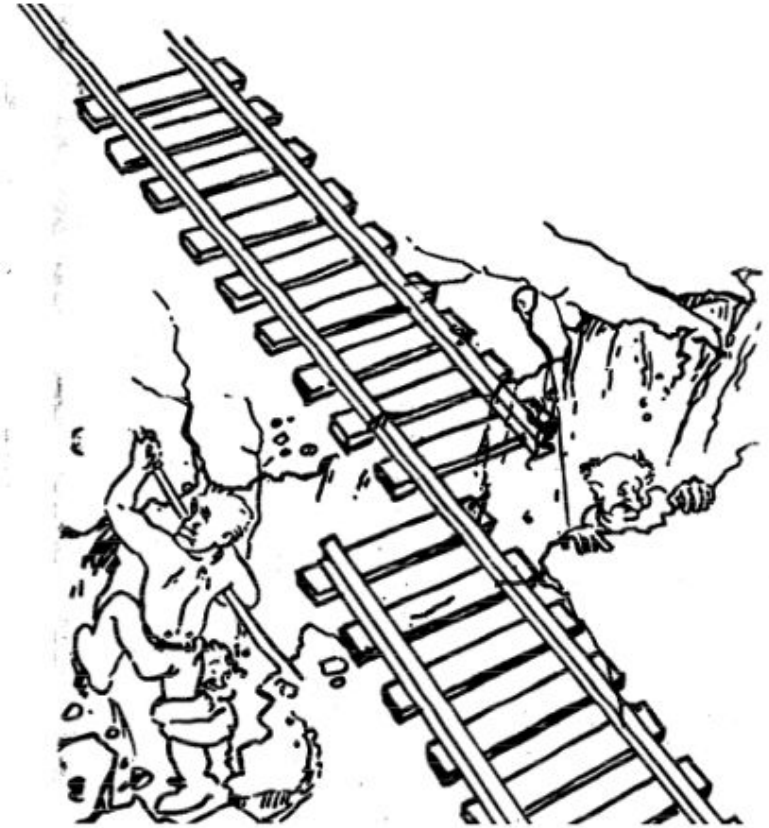
Difetto

- Problema di comunicazione tra team
- Implementazione errata di una specifica di uno dei team



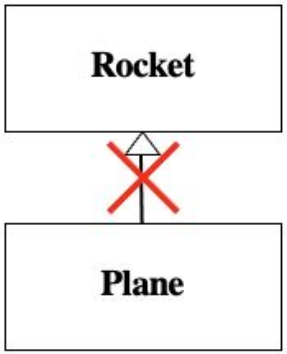
Difetto: la causa meccanica o algoritmica di un errore

Difetto meccanico



Bug F-16

- Dov'è il guasto?
- Dov'è l'errore?
- Qual è il difetto?
  - Cattivo uso dell'ereditarietà
  - Un aereo **non** è un razzo

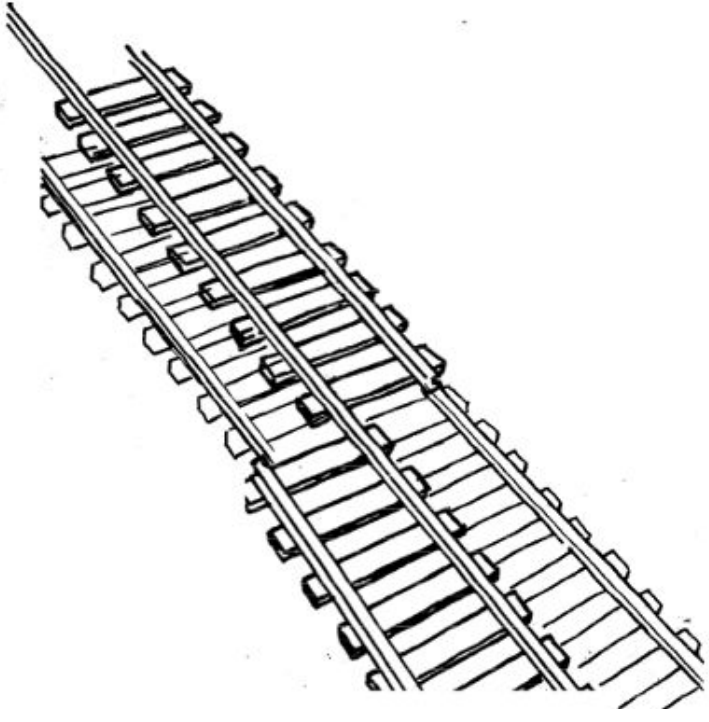


Esempi di difetti ed errori

- Difetti nella specifica dell'interfaccia
  - Differenza tra ciò che il client ha bisogno ed il server offre
  - Differenza tra requisiti ed implementazione
- Difetti algoritmici
  - Mancanza di inizializzazione
  - Condizione di branching sbagliata
  - Mancanza del test su null
- Difetti meccanici
  - Temperatura di funzionamento al di fuori delle specifiche dell'apparecchiatura
- Errori
  - Input utente sbagliato
  - Errori di riferimento a null
  - Errori di concorrenza
  - Eccezioni

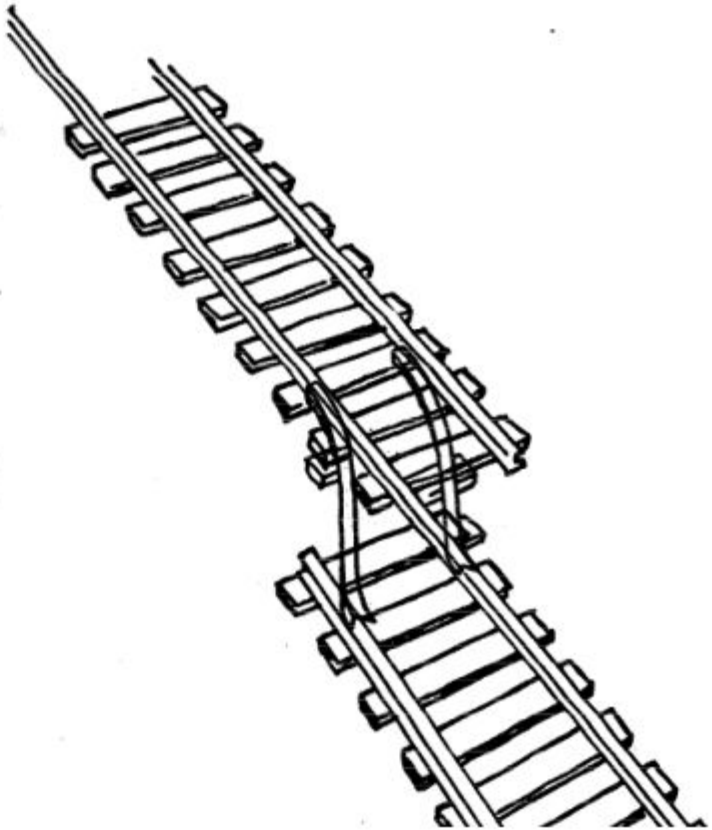
Come gestiamo errori, guasti e difetti?

- Ridondanza modulare



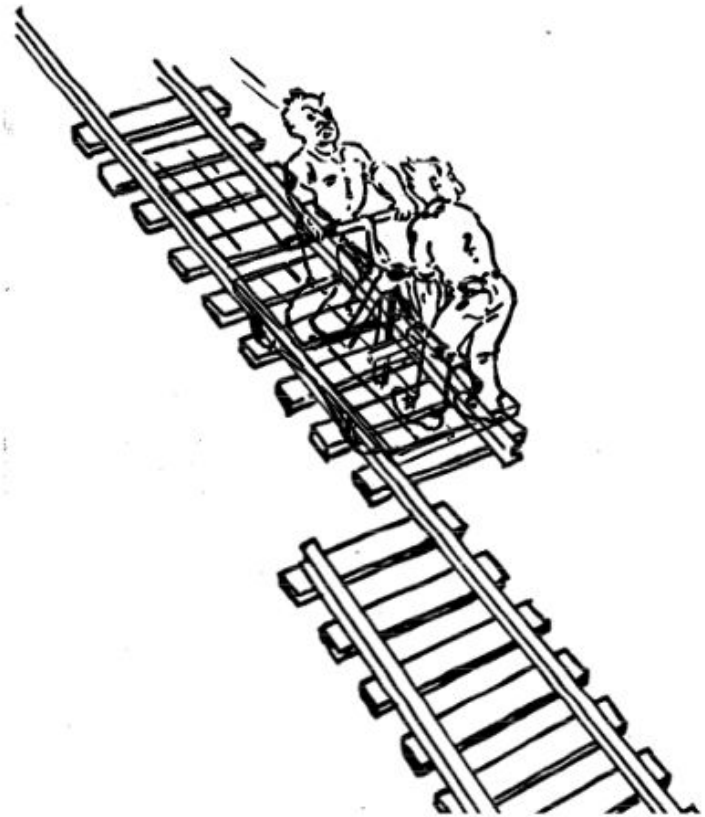
Come gestiamo errori, guasti e difetti?

- Patching



Come gestiamo errori, guasti e difetti?

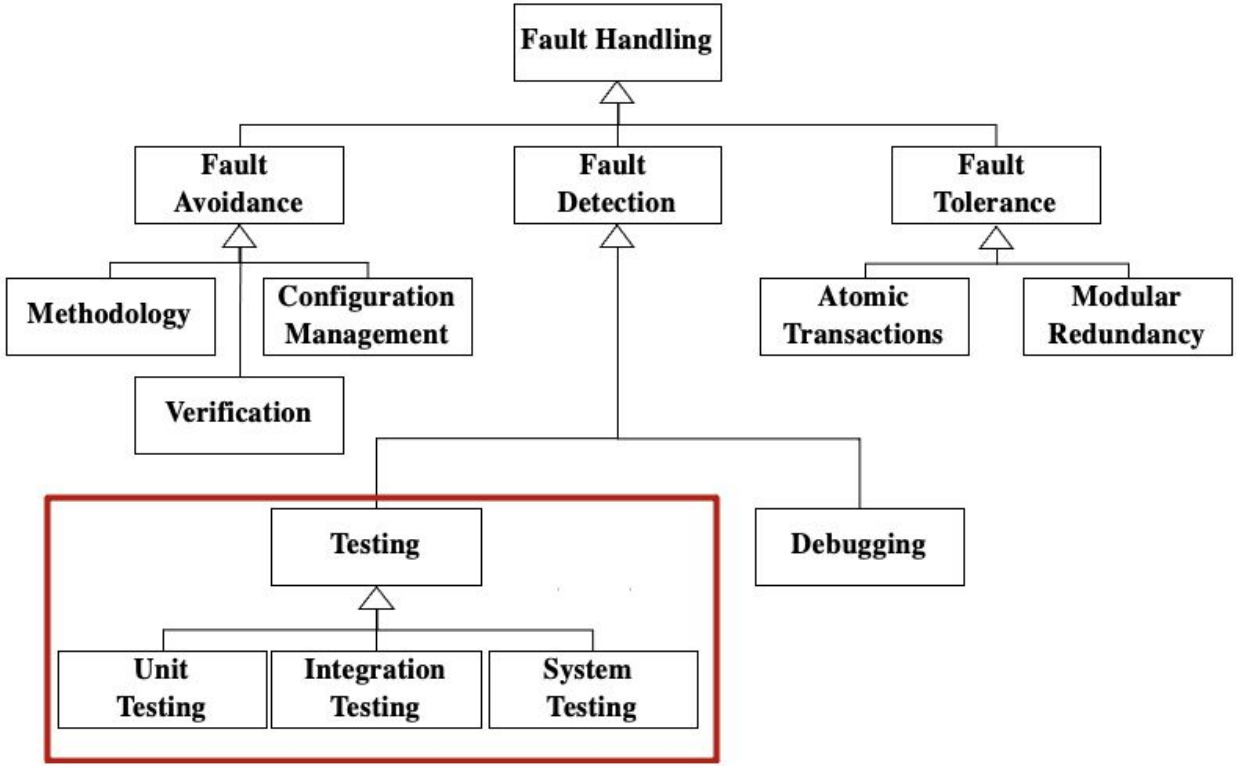
- Testing



Come trattare i difetti...

- Evitare i difetti (prima che il sistema sia rilasciato)
  - Usare metodologie per ridurre la complessità
  - Applicare la verifica per prevenire i difetti algoritmici
  - Usare le revisioni per identificare i difetti già nella progettazione
- Individuazione dei difetti (mentre il sistema è in esecuzione)
  - Testing: attività per provocare i guasti in modo pianificato
  - Debugging: trovare ed eliminare la causa (difetto) di un guasto osservato
  - Monitoraggio: fornire informazioni sullo stato e il comportamento -> usato durante il debugging
- Tolleranza ai guasti (ripristino dopo un rilascio del sistema)
  - Gestione eccezioni
  - Ridondanza modulare

Tassonomia delle tecniche di gestione dei difetti



Osservazioni

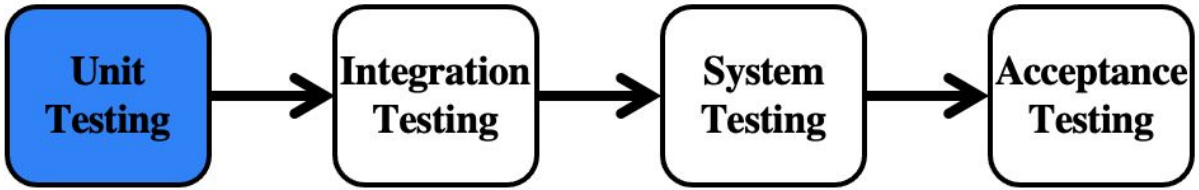
- E’ impossibile testare completamente un qualsiasi sistema o modulo non banale
  - Limitazioni pratiche: testing completo proibitivo per tempi e costi
  - Limitazioni teoriche: esempio, **problema della fermata**
- *‘Il testing può solo mostrare la presenza di bug, non la loro assenza’* **(Dijkstra)**
- Il testing non è gratis
  - Definire i propri obiettivi e priorità

Il testing richiede creatività

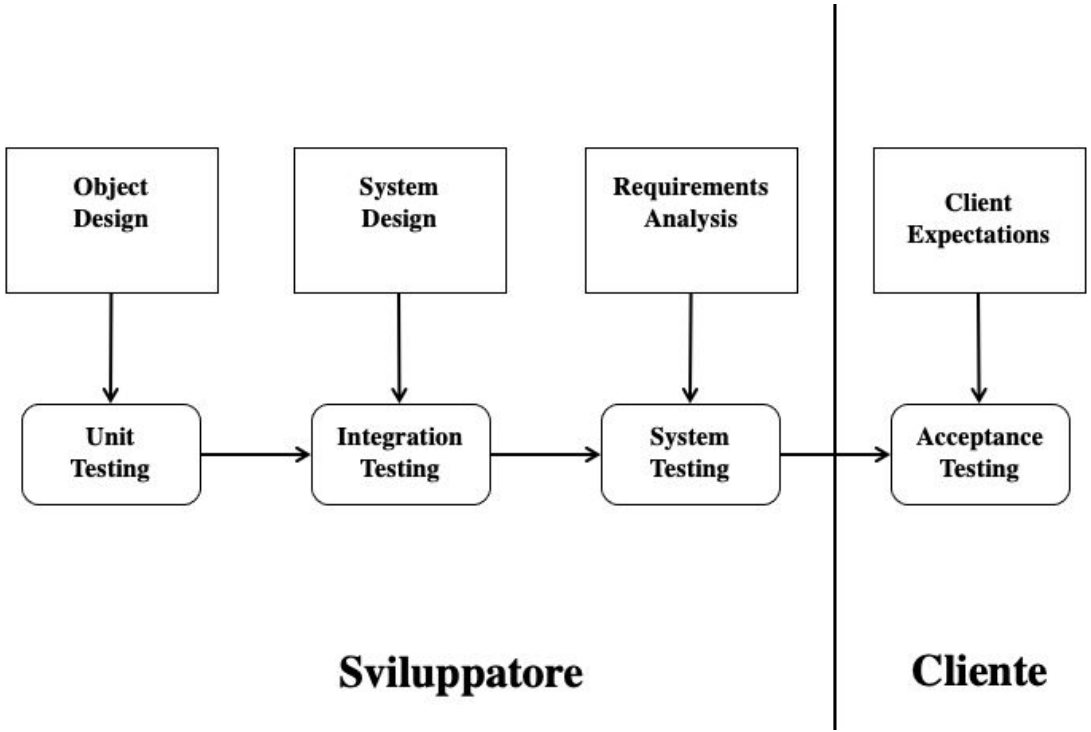
- Per sviluppare un test efficace, bisogna avere:
  - Una comprensione dettagliata del sistema
  - Conoscenza dei domini dell'applicazione e della soluzione
  - Conoscenza delle tecniche di testing
  - Abilità nell'applicare tali tecniche
- Il testing è realizzato meglio da personale di testing indipendente
  - Spesso sviluppiamo una certa attitudine mentale che il programma dovrebbe comportarsi in una certa maniera quando in effetti non lo fa
  - I programmatori spesso si attengono al set di dati che fa funzionare il programma
  - Un programma spesso non funziona quando provato da qualcun altro

Attività di test

- Testing unità
- Testing integrazione
  - Strategie di testing
- Testing di sistema
  - Testing funzionale
  - Testing di accettazione



Attività di testing e modelli



Tipi di test

- Testing delle unità
  - Sono testati singoli componenti (classi o sottosistemi)
  - Realizzato dagli sviluppatori
  - **Obiettivo**: confermare che il componente o sottosistema è programmato correttamente e realizza le funzionalità previste
- Testing di integrazione
  - Sono testati gruppi di sottosistemi e alla fine l'intero sistema
  - Realizzato dagli sviluppatori
  - **Obiettivo**: test delle interfacce tra i sottosistemi
- Testing del sistema
  - E' testato l'intero sistema
  - Realizzato dagli sviluppatori
  - **Obiettivo**: determinare se il sistema soddisfa i requisiti (funzionali e non funzionali)
- Testing di accettazione
  - Valuta il sistema consegnato dagli sviluppatori
  - Realizzato dal cliente. Può comportare l'esecuzione di transazioni tipiche in loco sulla base di prove
  - **Obiettivo**: dimostrare che il sistema soddisfi i requisiti e che sia pronto per l'uso

Quando dovrebbe essere preparato un test?

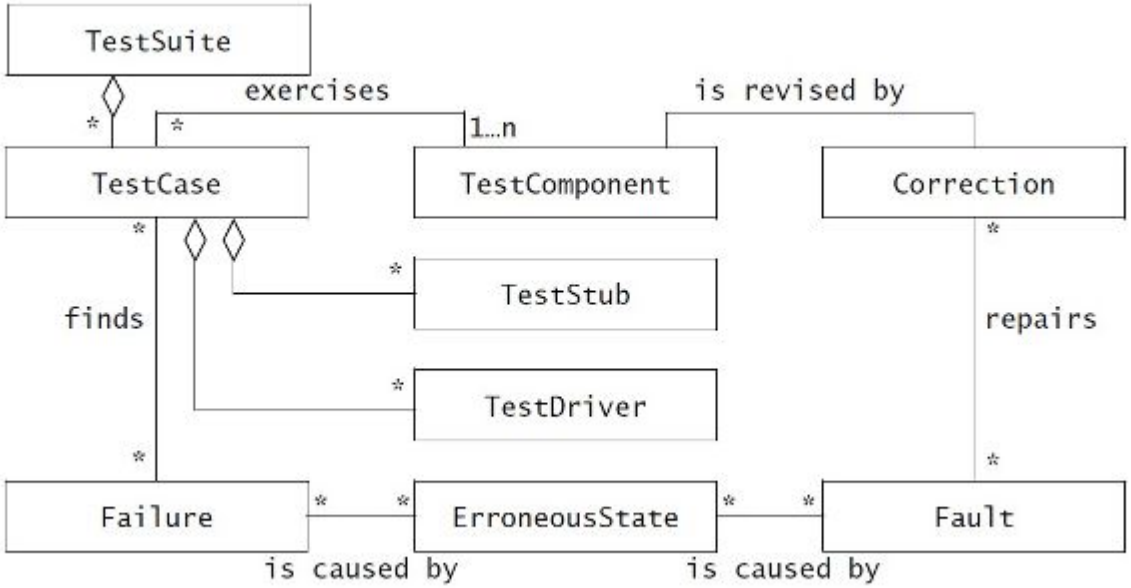
- Tradizionalmente dopo la stesura del codice sorgente

- Nella XP prima che sia scritto il codice sorgente
  - Ciclo di sviluppo guidato da test
    - Aggiungere un test
    - Eseguire i test automatici
      - Osservare il fallimento del nuovo test
    - Scrivere del codice
    - Eseguire i test automatici
      - Osservare il buon esito
    - Refactoring del codice

Terminologia

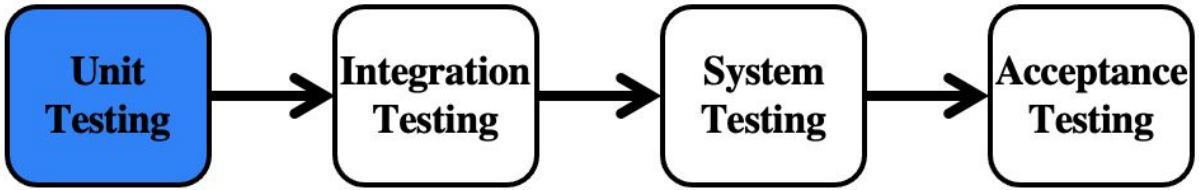
- Un componente di test è una parte del sistema che può essere isolato a scopo di test (oggetto, gruppo di oggetti, sottosistema)
- Un caso di test è un insieme di input e risultati attesi che esercitano un componente di test con lo scopo di causare guasti e individuare difetti
- Uno stub di test è un'implementazione parziale dei componenti da cui dipende il componente testato
- Un driver di test è un'implementazione parziale di un componente che dipende dal componente di test
- Gli stub di test e i driver permettono ai componenti di essere isolati dal resto del sistema per il testing
- Una correzione è una modifica ad un componente per riparare un difetto
  - Può introdurre nuovi difetti

Elementi usati durante il testing



Testing delle unità

- Testing statico (tempo di compilazione)
  - Analisi statica
  - Revisione
    - Walk-through (informale)
    - Ispezione del codice (formale)
- Testing dinamico (tempo di esecuzione)
  - Testing black-box
  - Testing white-box

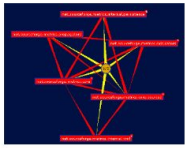


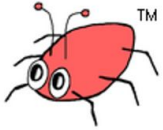
Analisi statica vs analisi dinamica

- Analisi statica
  - Esecuzione a mano: lettura del codice sorgente
  - Walk-through (presentazione informale ad altri)
  - Ispezione del codice (presentazione formale ad altri)
  - Controllo con strumenti automatici per
    - Errori sintattici e semantici
- Analisi dinamica
  - Testing black-box (test del comportamento input/output)
  - Testing white-box (test della logica interna del sottosistema o classe)
  - Testing basato su strutture dati (i tipi di dati determinano i casi di test)

Strumenti di analisi statica in Eclipse

- Warning ed errori del compilatore
  - Possibile variabile non inizializzata
  - Blocco vuoto non documentato
  - Assegnamento senza effetto
  - Punto e virgola mancante, …
- Checkstyle
  - Controlla le violazioni delle line guida per la codifica
  - http://checkstyle.sourceforge.net
- Metrics
  - Controlla anomalie strutturali
  - http://metrics.sourceforge.net
- FindBugs
  - Usa l'analisi statica per cercare bug in codice Java
  - http://findbugs.sourceforge.net





Osservazioni sull'analisi statica

- L'analisi statica normalmente trova gli errori ma alcuni errori non interessano
  - Importante trovare l'intersezione tra errori stupidi e importanti
- Non una bacchetta magica ma se usato in modo efficace, l'analisi statica è più economica di altre tecniche per intercettare gli stessi bug
- L'analisi statica, al più, in grado di determinare il 5-10% dei problemi di qualità del software
- Fonte: William Pugh, Mistakes that Matter, JavaOne Conference
  - http://www.cs.umd.edu/~pugh/MistakesThatMatter.pdf

Testing black-box

- Focus: comportamento I/O
  - Se per un qualsiasi input dato, possiamo predire l'output, allora l'unità passa il test
  - Richiede un *oracolo* di test
  - Quasi sempre impossibile generare tutti i possibili input (casi di test)
- Obiettivo: ridurre il numero di casi di test con le partizioni di equivalenza
  - Divide gli input in classi di equivalenza
  - Sceglie i casi di test per ogni classe di equivalenza
    - Esempio: se un oggetto si suppone che accetti un numero negativo, testare un numero negativo è sufficiente

Testing black-box: selezione casi di test

- Selezione delle classi di equivalenza (nessuna regola, solo linee guida)
  - L'input è valido attraverso un range di valori. Selezionare i casi di test da tre classi di equivalenza
    - Sotto il range
    - Nel range
    - Sopra il range
  - L'input è valido se è in un insieme discreto. Selezionare i casi di test da due classi di equivalenza
    - Valori discreti validi
    - Valori discreti non validi
- Altra soluzione per selezionare solo un numero limitato di casi di test
  - Sfruttare la conoscenza su dettagli interni dell'unità da testare -> testing white-box

Testing black-box: esempio

```
public class MyCalendar {

        public int getNumDaysInMonth(int month, int year)
                throws InvalidMonthException
        { … }
}
```

- Consideriamo le seguenti rappresentazioni

  `Month: (1,2,3,4,5,6,7,8,9,10,11,12)`

  dove 1= Jan, 2 = Feb, …, 12 = Dec

  `Year: (1904,…,1999,2000,…,2010)`

- Quanti casi di test sono necessari per fare un test di unità black-box completo di `getNumDaysInMonth()`?

Testing black-box: esempio

- Dipende dal calendario. Assumiamo il calendario gregoriano
- Classi di equivalenza per il parametro Month
  - Mesi con 30 giorni, mesi con 31 giorni, Febbraio e mesi illegali, 0, 13, -1
- Classi di equivalenza per il parametro Year
  - Un anno normale
  - Anni bisestili
    - Divisibile per 4
    - Divisibili per 100
    - Divisibili per 400
  - Anni prima e dopo Cristo
  - Anni illegali: prima del 1904, dopo il 2010
- Quanti casi di test sono necessari per fare un test di unita black-box completo di `getNumDaysInMonth()`?
  - **12 casi di test**

Testing white-box

- Focus: completezza (copertura). Ogni istruzione del componente è eseguita almeno una volta
- Quattro tipi di testing white-box
  - Testing istruzione (statement)
  - Testing del ciclo (loop)
  - Testing del percorso (path)
  - Testing delle condizioni (branch)

Testing white-box

- Testing di istruzione
  - Testa ogni istruzione
- Testing di ciclo
  - Ciclo da eseguire esattamente una volta
  - Ciclo da eseguire più di una volta
  - Salto completo della condizione del ciclo
- Testing di percorso
  - Assicura che tutti i percorsi del programma siano eseguiti
- Testing condizionale
  - Assicura che ogni esito in una condizione sia testato almeno una volta
  - Esempio:
    - Quanti casi di test sono necessari per il test di unità di questa istruzione?

```
if ( i = TRUE) printf("Yes");else  printf("No");
```

Esempio di testing condizionale

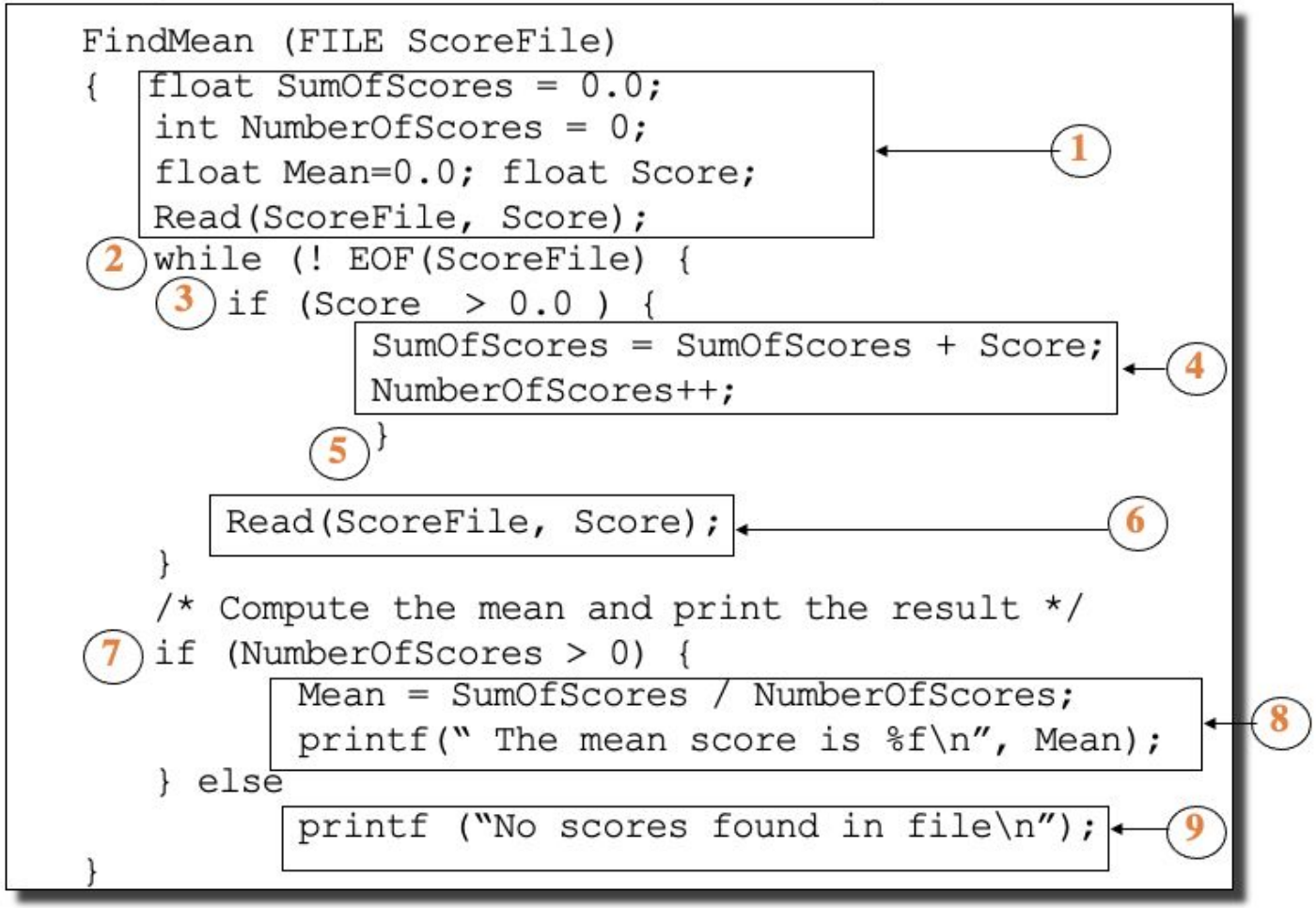
```
if ( i = TRUE) printf("Yes");else  printf("No");
```

- Abbiamo bisogno di due casi di test con i seguenti dati di input
  1. i = TRUE
  2. i= FALSE
- Qual è l'output atteso per i due casi?
  - In ambo i casi: Yes
  - Questo è un tipico errore da principianti nei linguaggi, dove l'operatore di assegnamento restituisce anche il valore assegnato (C, Java)
- Per cui anche i test possono essere difettosi
- Alcuni di tali difetti possono essere identificati con l'analisi statica

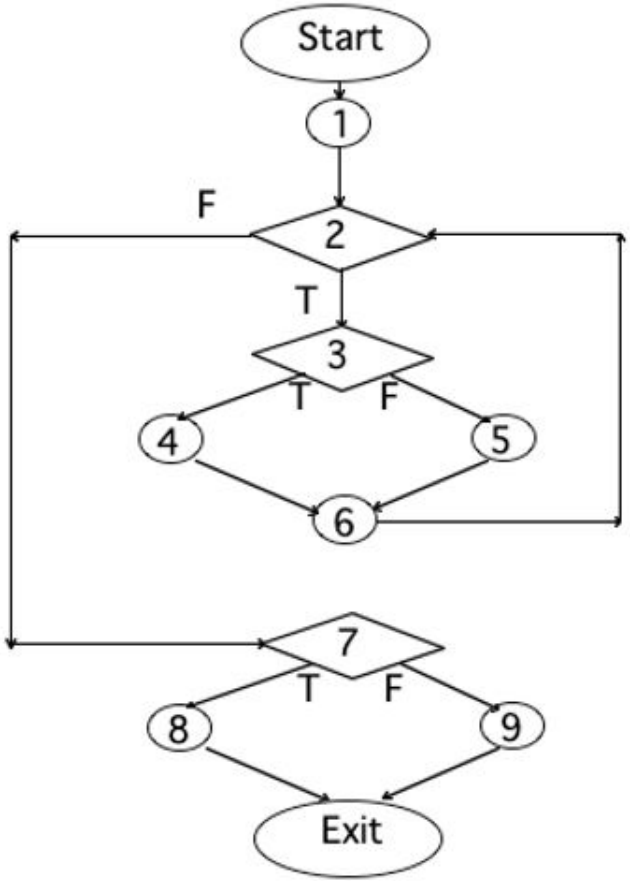
Esempio di testing white-box

```
FindMean(float Mean, FILE ScoreFile)
{ SumOfScores = 0.0; NumberOfScores = 0; Mean = 0;
 Read(ScoreFile, Score);  /*Read in and sum the scores*/
 while (! EOF(ScoreFile) {
         if ( Score > 0.0 ) {
               SumOfScores = SumOfScores + Score;
              NumberOfScores++;
          }
          Read(ScoreFile, Score);
 }
 /* Compute the mean and print the result */
 if (NumberOfScores > 0 ) {
              Mean = SumOfScores/NumberOfScores;
              printf("The mean score is %f \n",  Mean);
 } else
              printf("No scores found in file\n");
}
```

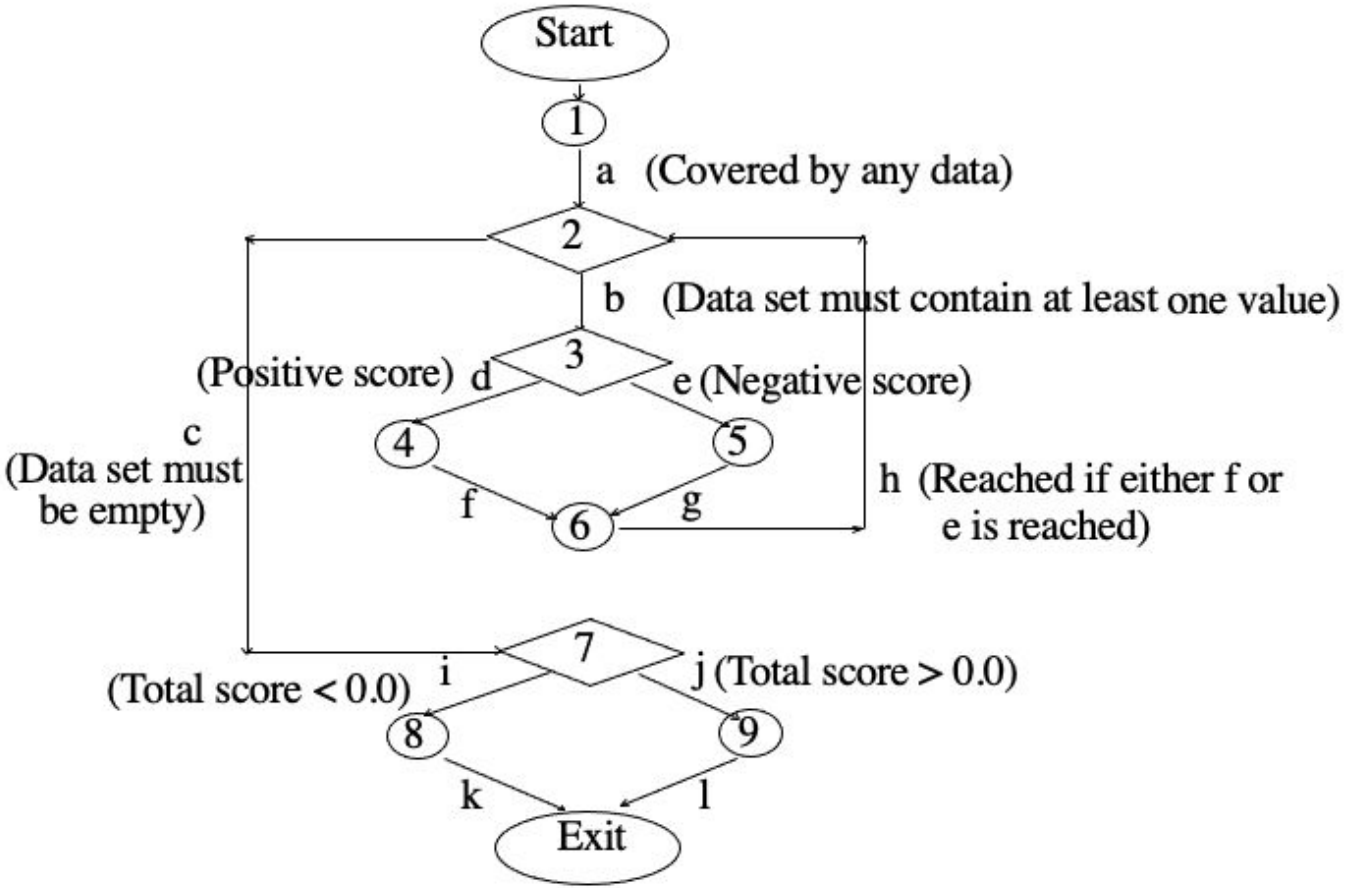
Testing White-box: determinare i percorsi



Costruire il diagramma del flusso logico



Trovare i casi di test



Casi di test

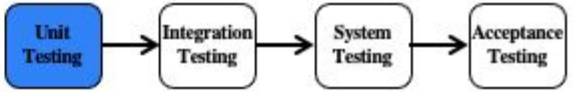
- Caso di Test 1 : ? (da eseguire il ciclo esattamente un a volta)
- Caso di Test 2 : ? (saltare completamente il ciclo)
- Caso di Test 3: ?,? (eseguire il ciclo più di un volta)

Questi tre casi di test coprono tutti i percorsi del flusso di controllo

Euristiche per il testing di unità

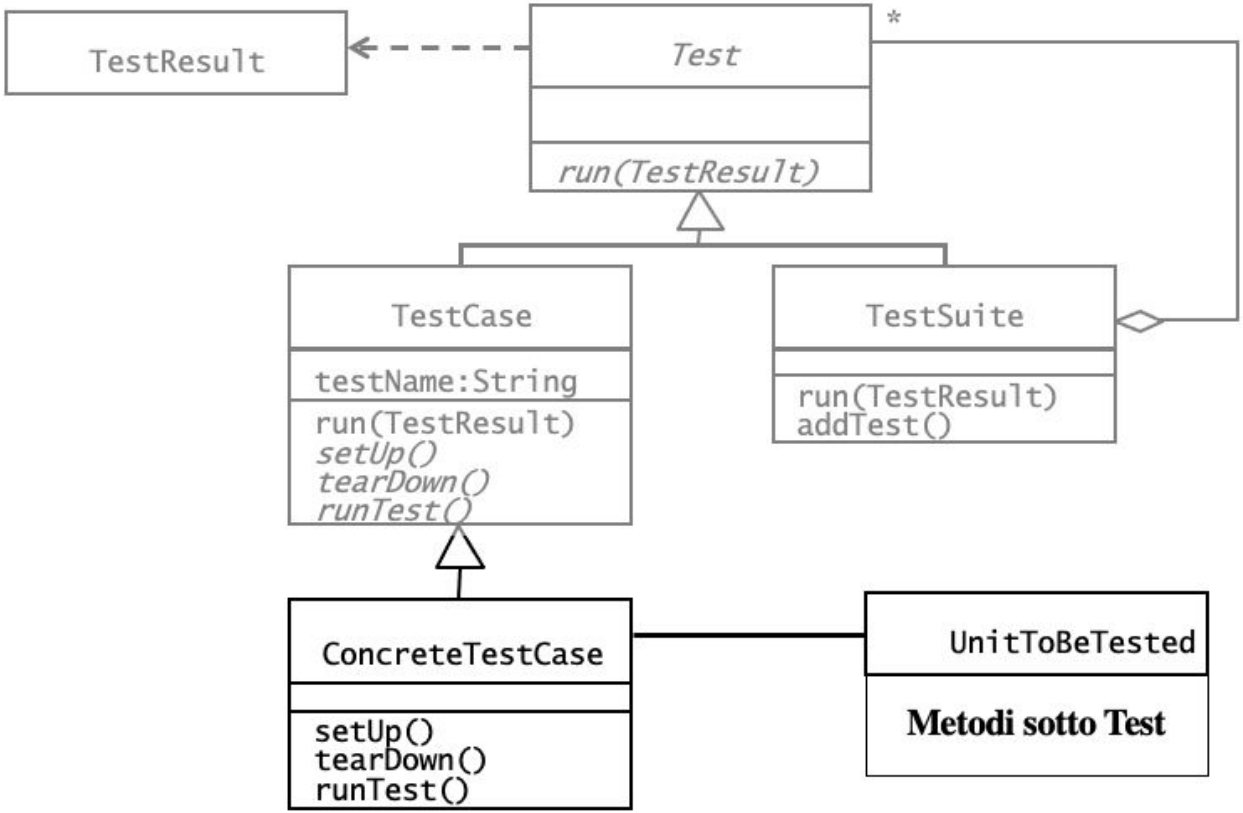
1. Creare i test di unità quando è completato il progetto degli oggetti
   - Test black-box: test del modello funzionale
   - Test white-box: test del modello dinamico
2. Sviluppare i casi di test
   - Obiettivo: trovare un numero efficace di casi di test
3. Cross-check dei casi di test per eliminare duplicati
   - Non sprecare tempo!
4. Controllare a tavolino il proprio codice
   - Talvolta riduce il tempo per il testing
5. Creare un impianto (fixture) di test
   - I driver del test e gli stub sono necessari per il testing di integrazione
6. Descrivere l'oracolo del test
   - Spesso il risultato del primo test eseguito con successo
7. Eseguire i casi di test
   - Rieseguire il test quando è fatta una modifica (**test di regressione**)
8. Confrontare i risultati del test con l'oracolo del test
   - Automatizzarlo se possibile

JUnit: panoramica



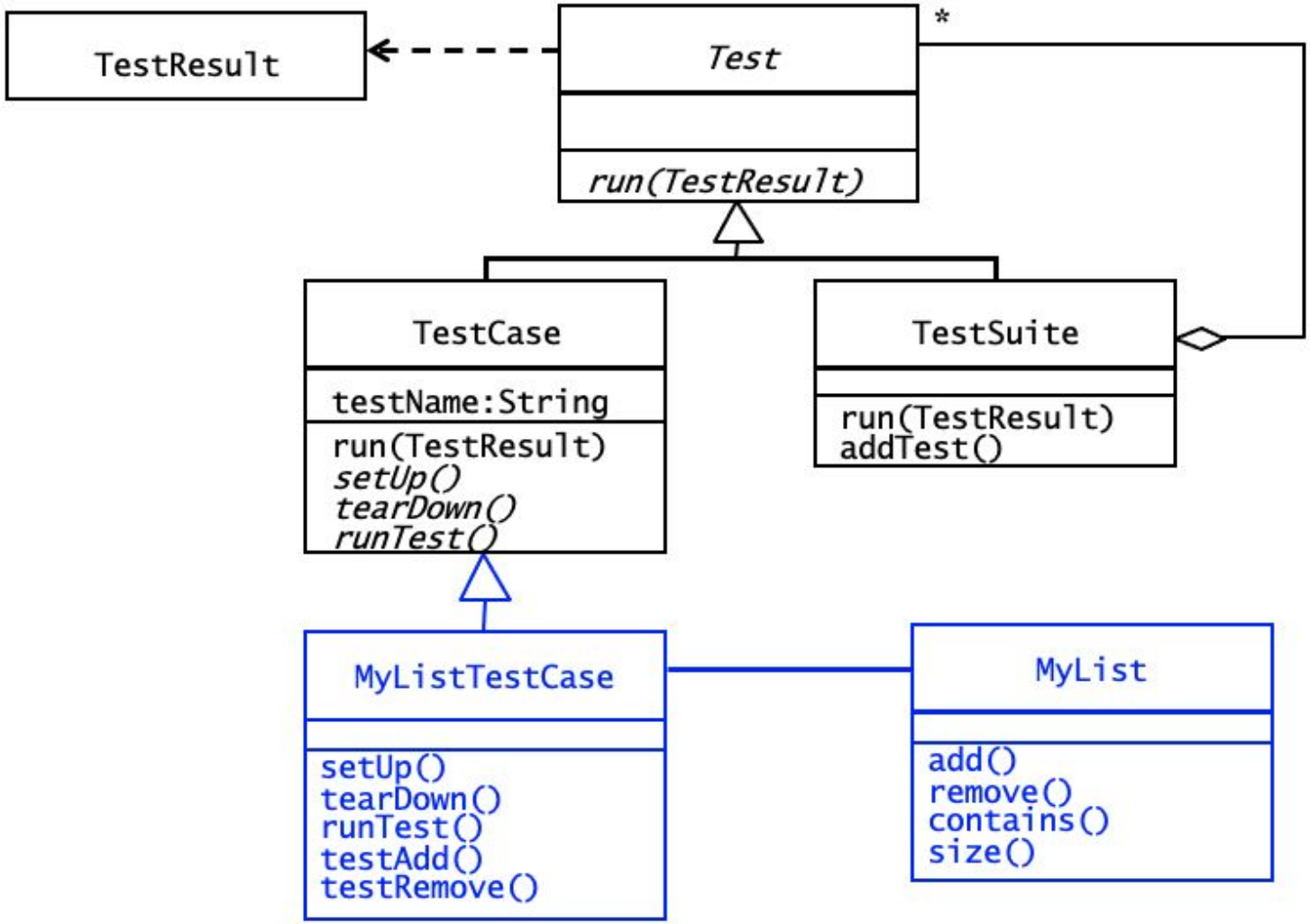
- Un framework Java per la scrittura e l'esecuzione di test di unità
  - Casi di test
  - Suite di Test
  - Esecutore test
- Scritto da Kent Beck e Erich Gamma
- Scritto con l'idea del "testare prima" e sviluppo basato su pattern
  - I test scritti prima del codice
  - Permette il testing di regressione
  - Semplifica il refactoring
- JUnit è Open Source
  - www.junit.org

Classi JUnit



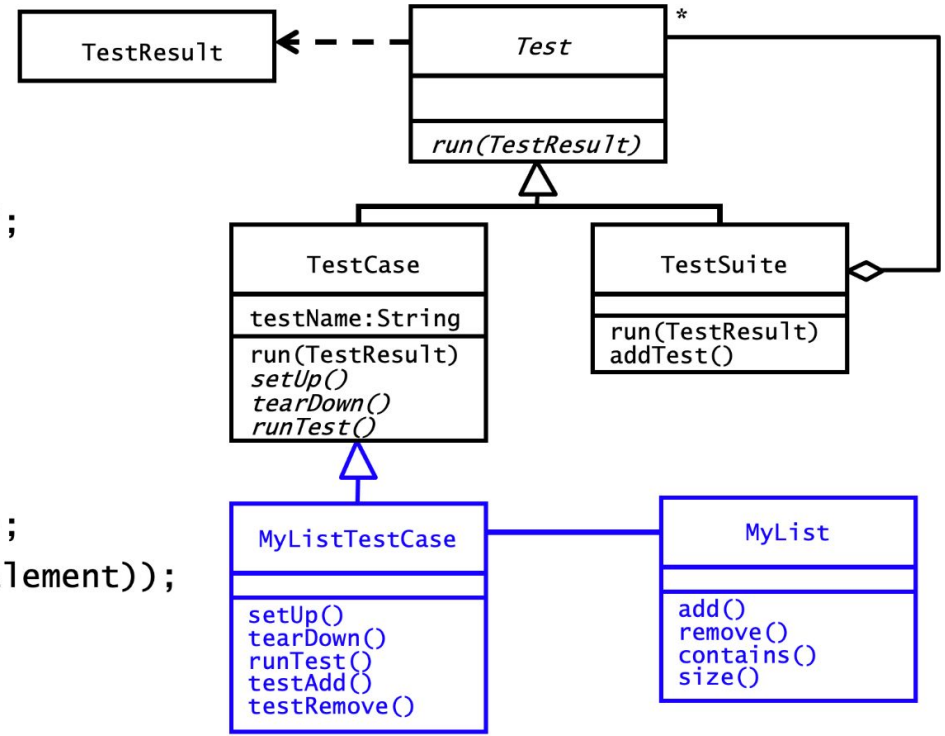
Esempio: Testing MyList

- Unità da testare
  - MyList
- Metodi sotto test
  - add()
  - remove()
  - contains()
  - size()
- Caso di test concreto
  - MyListTestCase

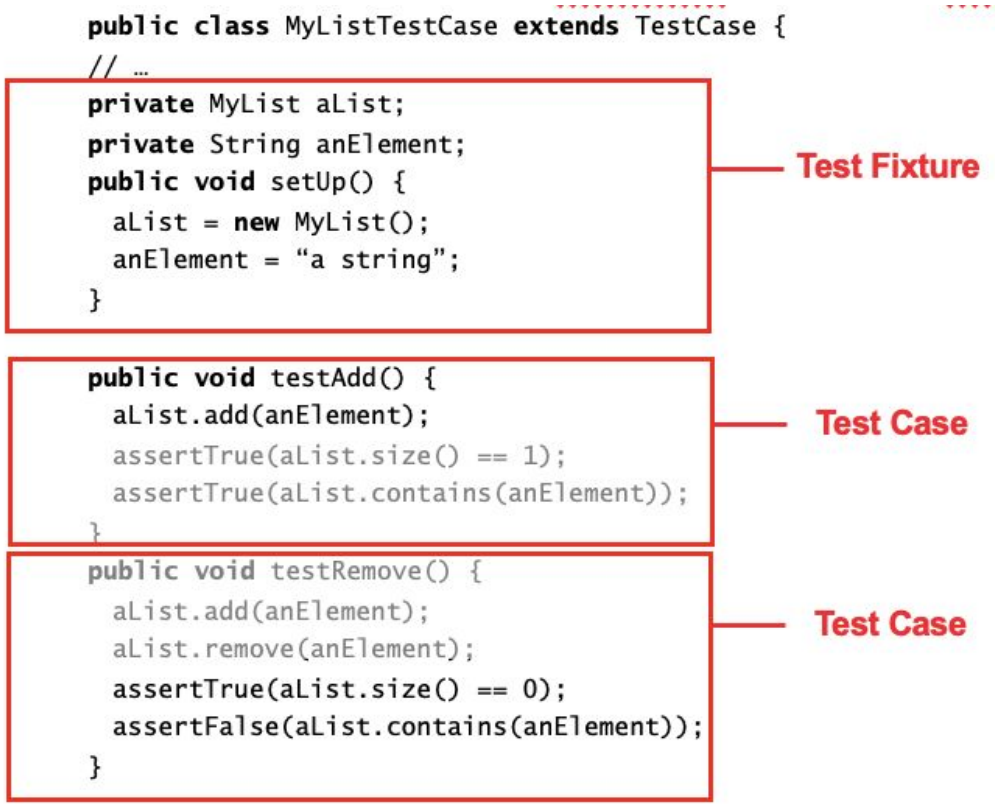

TestResult
Test
*
run(TestResult)
TestCase
testName:String
run(TestResult)
setUp()
tearDown()
runTest()
TestSuite
run(TestResult)
addTest()
MyListTestCase
setUp()
tearDown()
runTest()
testAdd()
testRemove()
MyList
add()
remove()
contains()
size()

TestCases in JUnit

```
public class MyListTestCase extends TestCase {

public MyListTestCase(String name) {
  super(name);
}
public void testAdd() {
   // Set up the test
  List aList = new MyList();
  String anElement = "a string";

   // Perform the test
  aList.add(anElement);

   // Check if test succeeded
  assertTrue(aList.size() == 1);
  assertTrue(aList.contains(anElement));
}
protected void runTest() {
  testAdd();
}
}
```



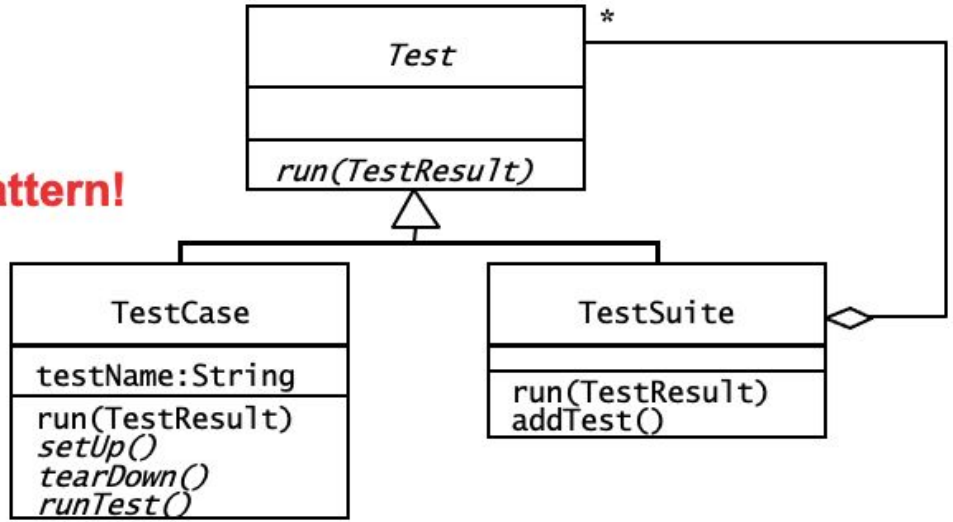
Scrivere Fixture e casi di test



Collezionare casi di test in suite di test

```
public static Test suite() {
 TestSuite suite = new TestSuite();
 suite.addTest(new MyListTest("testAdd"));
 suite.addTest(new MyListTest("testRemove"));
 return suite;
}
```

Composite Pattern!



Confronto dei testing white box e black-box

- Testing white-box
  - Deve essere testato un numero potenzialmente infinito di percorsi
  - Spesso testa cosa è fatto, invece di cosa dovrebbe essere fatto
  - Non può individuare casi d'uso mancanti
- Testing black-box
  - Potenziale esplosione combinatoria dei casi di test (dati validi e non)
  - Spesso non chiaro se i casi di test selezionati scoprono un errore particolare

- Necessari ambo i tipi di testing
  - Testing White e black-box sono gli estremi di uno stesso spettro continuo di testing
  - Qualsiasi scelta di caso di test si trova nel mezzo e dipende da:
    - Numero di possibili percorsi logici
    - Natura dei dati di input
    - Quantità di computazione
    - Complessità di algoritmi e strutture dati