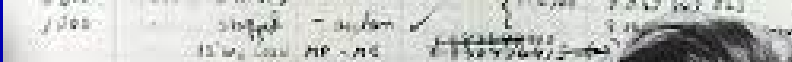
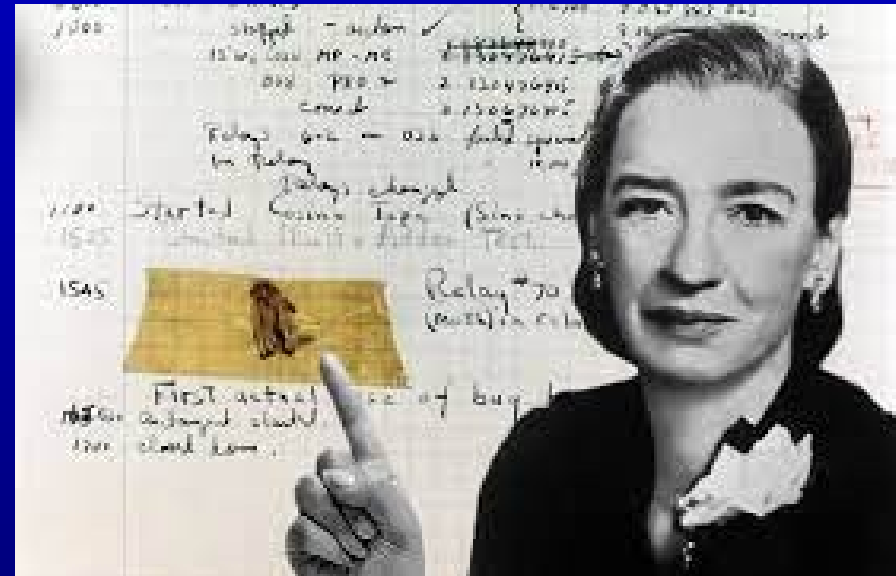
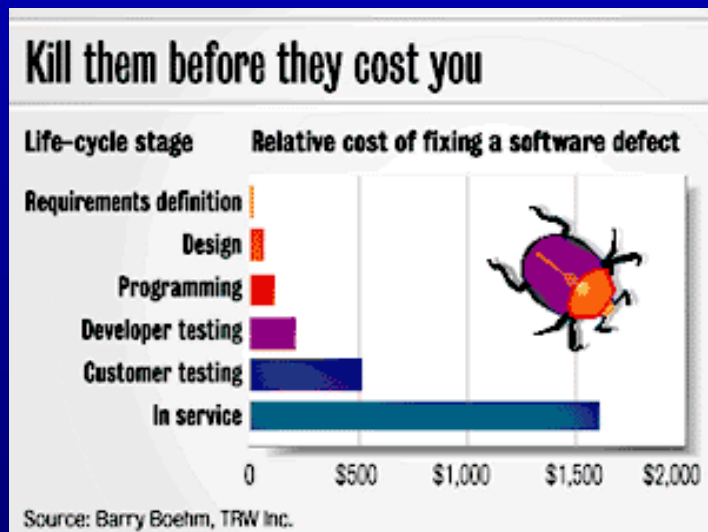


# Testing del software

# Testing del software

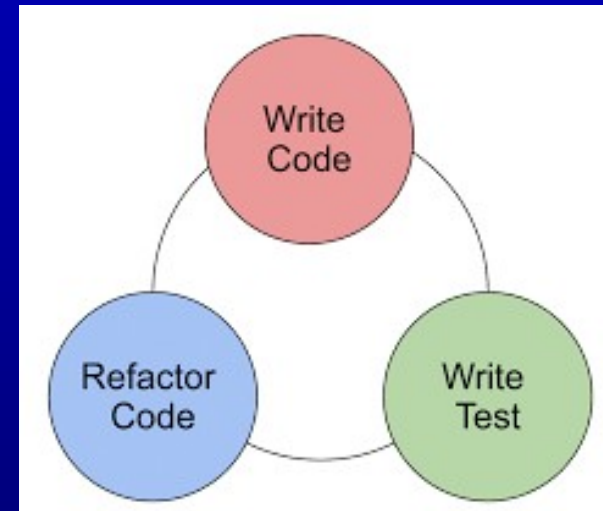
- *Necessità di testing - verifica della correttezza del software*
  - Il processo di testing è **concettualmente semplice, ma lungo e tedioso**
    - È soggetto ad errori umani
    - Non scala con il codice
    - *Va automatizzato!*
- 



# Benefici del testing

Molto più dell'eliminazione dei malfunzionamenti

- **Sviluppo incrementale** facilitato
- **Miglioramento del design** di un software
  - Spinge verso la modularità - testing delle varie parti
- **Forma di documentazione del codice**
- **Facilitazione del refactoring**
  - Durante la riorganizzazione del codice si procede per piccole trasformazioni e relativi test, per poter tornare in uno stato consistente



# Importanza del testing (ben fatto)

- *Testing necessario e fondamentale*
- *Presenza di team dedicati a sole attività di testing*
- Il testing di un software di dimensioni medio-grandi rappresenta un compito tutt'altro che semplice
- Nella storia, le conseguenze di un errore a run time non identificato dal testing spaziano *dal ridicolo al catastrofico...*

# Ariane 5

**Lanciatore dell'ESA**

**4 giugno 1996: il razzo si autodistrugge dopo 40 secondi dal lancio per via di un**

**malfunzionamento del software di controllo**

**Un dato a 64 bit in virgola mobile, che rappresentava la velocità orizzontale rispetto alla piattaforma di lancio, venne convertito in un intero a 16 bit con segno: il numero in virgola mobile era troppo grande per poter essere rappresentato con un intero a 16 bit → trap del processore**

**Fu necessario quasi un anno e mezzo per capire quale fosse stato il malfunzionamento che aveva portato alla distruzione del razzo.**



# Sonda Mars Climate Orbiter

**23 settembre 1999: la sonda Mars Climate Orbiter si abbatte sul suolo marziano a causa di un “silly mistake”: gruppi di sviluppo del software di bordo e del software di terra, appartenenti a paesi differenti, utilizzavano diverse unità di misura (libbra / forza del sistema imperiale britannico vs newton del sistema metrico decimale)**

**Costo totale: 328 milioni di dollari**



# Tipologie di test

- **Functional (Unit) test:** (molto frequente)
  - Si verificano le funzionalità di singoli moduli/parti
- **Integration (acceptance) test:** (molto frequente)
  - Si verificano le funzionalità di alto livello del programma, di solito utilizzate dal cliente finale
- **Regression test:** (frequente in progetti grandi)
  - Si verifica che le modifiche introdotte (es. patch, cambi di configurazione, bug fixing) non portino ad una *regressione*
  - Ricerca di *bug* “storici” (piuttosto comune)
- **Performance (stress) test:** (ambiente server)
  - Si verifica il livello di prestazione del software

# Tecniche più usate

## Black-box testing

- **Equivalence partitioning**
  - Divisione dei dati di input in partizioni (range)
- **Boundary value analysis**
  - Valori rappresentativi degli estremi delle partizioni
- **Fuzz testing**
  - Dati random, inaspettati (es. tipi non validi)

## White-box testing

- **Static testing**
- **Code coverage**



# Realizzare uno Unit Test completo

Si parte da un programma già suddiviso in moduli

- Se non lo è, occorre modularizzarlo tramite un processo di refactoring
- Si testano **tutti** i moduli
- Per ogni modulo si testano **tutte** le funzioni, considerando due aspetti
  - **TUTTI** gli input validi e non validi – *testing Black box*
  - Gli input necessari ad attivare **TUTTI** i possibili percorsi di codice (**code path**) all'interno di ogni singola funzione – *testing White box*

# Esempio

Specifiche: il programma legge da stdin un numero intero  $n$  che deve essere compreso tra 1 e 5. Quindi legge  $n$  valori interi diversi da 0 ed una stringa (senza spazi). I valori interi letti sono inseriti nelle prime  $n$  posizioni di un array di 5 elementi. Se  $n < 5$ , i rimanenti elementi dell'array, ossia quelli di indice compreso tra  $n - 1$  e 4, sono riempiti con l'ultimo valore intero letto. Ad esempio, se si immette  $n=3$  ed i valori 7, -9, e 2, l'array conterrà i valori [7, -9, 2, 2, 2]. Per ciascuno elemento dell'array, il programma ristampa il valore assoluto dell'elemento in modulo con il valore assoluto della somma dei valori dei 5 elementi dell'array. Infine il programma ristampa la stringa.

Secondo il precedente esempio, la somma dei valori è 4, per cui, in quanto agli elementi dell'array, il programma ristampa:

$$7 \% 4 = 3$$

$$9 \% 4 = 1$$

$$2 \% 4 = 2$$

$$2 \% 4 = 2$$

$$2 \% 4 = 2$$

Infine, il programma deve terminare ritornando lo stato di uscita 0.

Quali controlli black-box fareste?

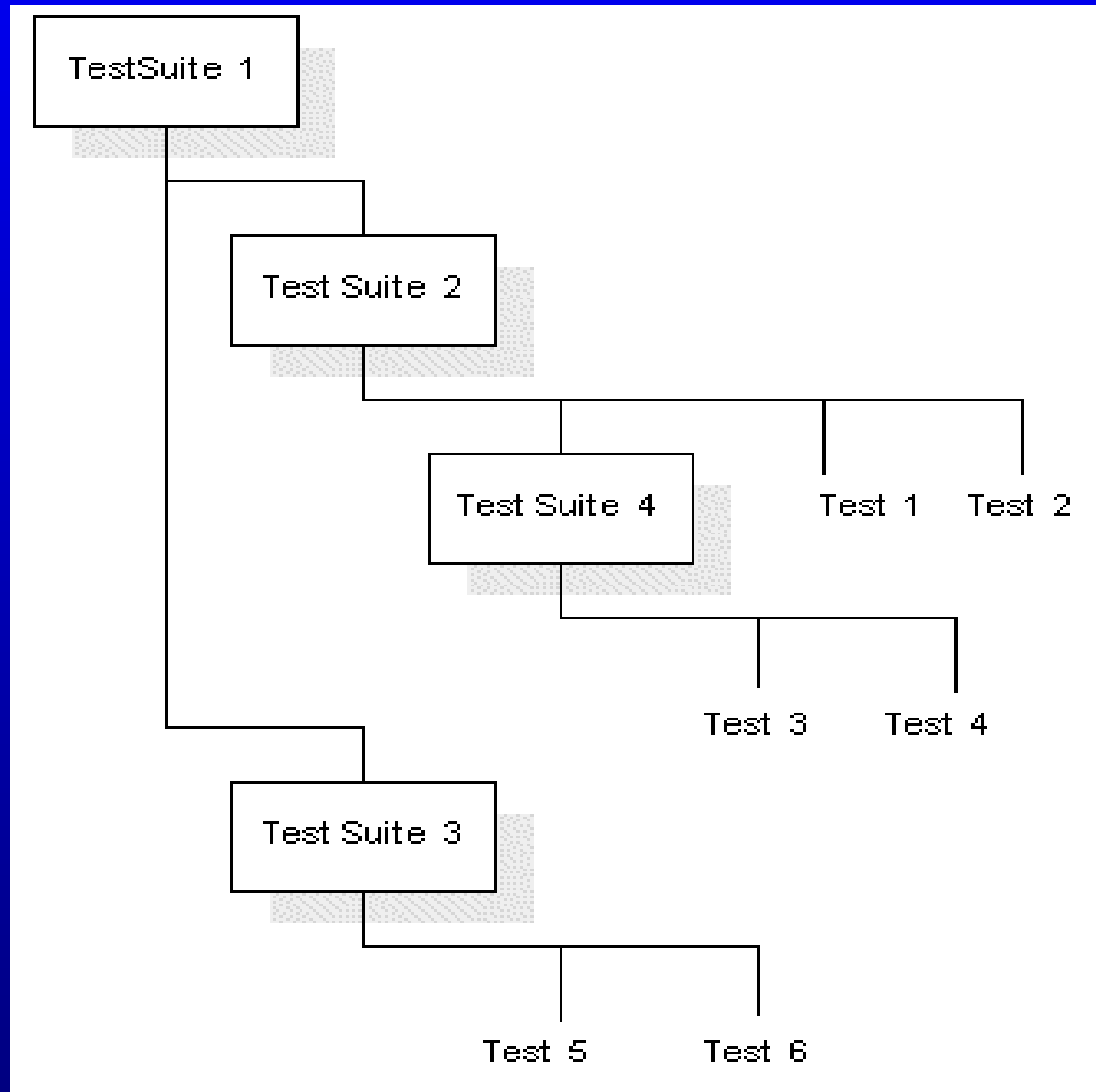
# Unit Testing

- Meccanismo per verificare automaticamente le funzionalità di una parte del software
- I building block di uno Unit Test sono i test case
- Ciascun test case è un insieme di funzioni di test e mira a testare uno specifico aspetto
- Più test case che condividono lo stesso obiettivo possono essere raggruppati in una test suite
  - Es. Obiettivo suite: testare il comportamento del software con input non validi – ciascun test case testa diversi tipi di input

Test di vari tipi: *consistenza architetturale, coerenza funzionale, robustezza, copertura del codice*

# Unit Testing

Struttura  
**gerarchica:**  
suite, sotto-suite  
e test\_case



# Funzionalità dello Unit Test

- Le **funzioni dei test case** non prendono in ingresso alcun parametro, né forniscono in uscita alcun valore
  - Input/parametri da testare già decisi/noti
  - Necessità di “rifare” le operazioni da testare
- All'interno di ciascuna funzione di test, si verifica se un **risultato di una operazione è coerente con un risultato atteso** (meccanismo basato su **asserzioni**)
- Alla fine di un test, viene stampato un **resoconto dettagliato**

# Assertzioni

- Una **asserzione** è una funzione che prende in ingresso alcuni **parametri**:
  - il risultato di un metodo o l'oggetto da testare (*obbligatorio*)
  - il risultato atteso di riferimento (*opzionale*)
  - Il messaggio stampato in caso di errore (*opz.*)
- L'asserzione verifica una **data proprietà** (ad es., uguaglianza fra l'oggetto da testare e il corrispondente riferimento)
  - Se la proprietà non vale (**test fallito**) viene prodotto un **report dettagliato**

# Asserzioni

- Esempi di asserzioni comuni:
  - **assert\_true(expr, failure message):** verità di una espressione booleana (expr)
  - **assert\_equal(expected, actual, msg):** uguaglianza di due espressioni numeriche (expected, actual)
  - **assert\_match(exp, act, msg):** match di due stringhe (exp, act)
  - **assert\_not\_nil(obj, msg):** oggetto non nullo (obj)
  - **assert\_raise(exception, msg) { codice}:** sollevamento di una eccezione (exception)

# Dettagli implementativi

- In molti linguaggi è presente (integrato) un **modulo** per lo Unit Testing
  - **Junit** (Java), **Test::Simple** (Perl), **unittest** (Python), **Test::Unit** (Ruby)
- Ciascun Test Case è implementato come una **classe che può avere diversi metodi di test**
  - La classe deriva da una **classe madre** Test Case che mette a disposizione meccanismi e metodi per l'**esecuzione automatica** dei test
  - I metodi di test eseguono al loro interno **operazioni ed asserzioni**



# Dettagli implementativi

- **Suite di test** implementata tramite una **classe specifica** messa a disposizione dal linguaggio
  - Sequenza di classi Test Case incluse nella suite
  - Metodi a disposizione per l'**esecuzione automatica** dell'intera suite
  - Metodi a disposizione per la **generazione di report riassuntivi** nei formati più disparati (testo, PDF, HTML)

# Metodi Setup e Teardown

- A volte vi è la necessità di **eseguire del codice** prima (o dopo) di procedere con l'esecuzione della serie di test veri e propri
  - Es: creazione di istanze di classi usate nella parte di programma da testare, creazione DB di prova, lancio di processi server, etc
- Test::Unit mette a disposizione i metodi:
  - **Setup**: eseguito prima di ciascun test (inizializzazione)
  - **Teardown**: eseguito dopo ciascun test (cleanup)

# Unit Testing in Python

# Componenti e nomenclatura

## Modulo Python unittest

- **Test case:** è l'unità di test più piccola possibile  
**TestCase class**
- **Setup:** operazioni di preparazione propedeutiche per un test  
**Metodi setUp e tearDown di TestCase**
- **Test suite:** collezione di test case e/o di test suite  
**TestSuite class**
- **Test runner:** componente che orchestra l'esecuzione dei test e fornisce un report (testuale, grafico) all'utente  
**TestRunner e sottoclassi (es. *TextTestRunner*),  
TestLoader**

# La classe TestCase

- Classe **TestCase**: contenitore di test
- **Unit Test**
  - sottoclasse di **TestCase**
  - contiene un metodo per ciascun test che si vuole effettuare
- I nomi dei metodi di test sono **arbitrari**, ma devono iniziare con le lettere “**test**” (es. *testNome*) per essere identificati ed eseguiti **automaticamente dall'interprete (reflection)**
- I metodi contengono asserzioni per verificare il comportamento del software

# Assertzioni Python più usate

- Alcune asserzioni (**msg**: messaggio di riconoscimento del test nella reportistica)
  - **assertTrue(expr, msg)**: verifica `expr == True`
  - **assertFalse(expr, msg)**: verifica `expr == False`
  - **assertEqual(a, b, msg)**: verifica `a == b`
  - **assertNotEqual(a, b, msg)**: `a != b`
  - **assertRaises(exception, callable, args)**: verifica che l'invocazione di *callable* con argomenti *args* sollevi una eccezione di tipo *exception* (possibile passare una *tupla di exception classes*)  
**Fallisce** se non viene sollevata alcuna eccezione  
**Dà errore** se viene sollevata una eccezione di tipo diverso da quella specificata

# Assertzioni Python più usate

- Assertzioni di quasi (non) uguaglianza
  - `assertAlmostEqual(first, second, msg, places=7, delta=None)`: verifica che `first` sia uguale a `second` fino a *places* cifre decimali (default = 7) oppure che la loro differenza sia inferiore a *delta*
    - Specificare *uno solo* dei due parametri (*places* e *delta*), *altrimenti* viene sollevata una eccezione `TypeError`
  - `assertAlmostNotEqual(first, second, msg, places=7, delta=None)`: verifica che `first` e `second` siano approssimativamente diversi

# Assertzioni Python più usate

- Altre asserzioni molto usate
  - `assertGreater(a, b)`: verifica a maggiore di b
  - `assertGreaterEqual(a, b)`: verifica a maggiore o uguale di b
  - `assertLess(a, b)` e `assertLessEqual(a, b)`: verifica a minore (o minore uguale) di b
  - `assertRegexMatches(text, regexp)`: verifica che regexp faccia match con text

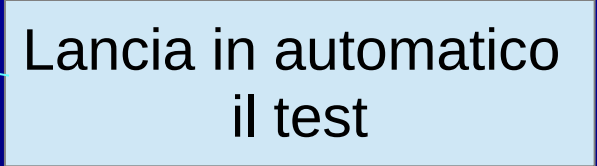
*Regexp = regular expression*
  - `assertNotRegexMatches(text, regexp)`: verifica che regexp non faccia match con text



# Esecuzione di uno Unit Test

- **Per creare ed eseguire uno Unit Test:**
  - Importare il modulo da verificare
  - importare il modulo Python **unittest**
  - creare una **sottoclasse della classe TestCase** che implementa il test specifico
  - Creare i **metodi testXXX** contenenti asserzioni
  - invocare il metodo **main()** del modulo **unittest**
- Il codice di invocazione di **main()** può essere integrato nel file contenente la **classe di test**

**unittest.main()**



Lancia in automatico  
il test

# Test Setup

- Frequente necessità di operare su risorse (oggetti) che devono essere preallocati
- **TestCase** fornisce funzioni per implementare il **meccanismo della TestFixture**
  - **invocazione automatica** dei metodi di allocazione e rilascio delle risorse
  - **Metodo setUp()**: allocazione di risorse (invocata prima del primo test)
  - **Metodo tearDown()**: rilascio di risorse (invocata dopo l'ultimo test)

# Risultato di un test

- Ciascun test ha tre possibili risultati:
  - **Ok**: esito positivo
  - **FAIL**: esito non positivo se solleva una eccezione **AssertionError** (una asserzione non è vera)
  - **ERROR**: esito non positivo se solleva una eccezione diversa da **AssertionError** (c'è un errore nel codice o un comportamento imprevisto)
    - Es. **assertRaises(exception, callable, args)** se rileva una eccezione diversa da quella specificata in **exception**

# Esecuzione di un test

```
# simpletest.py
```

```
import unittest
```

```
class SimplisticTest(unittest.TestCase):
```

Sottoclasse  
di unittest

```
    def testProva(self):
```

```
        self.assertEqual(1,3-2)
```

Metodo testXXX

```
if __name__ == '__main__':
```

```
    unittest.main()
```

Assertione X  
chiamata attraverso  
istanza **self.X**

Invocazione  
unittest.main()

```
$ python simplistictest.py
```

```
Ran 1 test in 0.001s
```

```
OK
```

Esecuzione test

Output

# Esempio di test

- Osserviamo la classe **Cerchio** ed esaminiamone gli attributi e i metodi forniti
- Ragioniamo su **quali test** realizzare nelle relative classi di Unit Test e su come **raggrupparli logicamente nei metodi** di test
- Test di *coerenza architetturale, coerenza funzionale (metodi invocabili, correttezza risultati dei metodi), robustezza (input invalidi)*
- Utilizzo della funzione **setUp()** per la creazione di una istanza della classe con parametri noti da testare