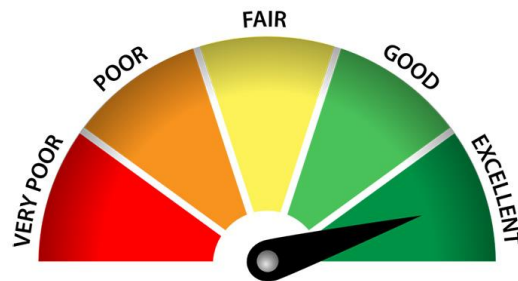


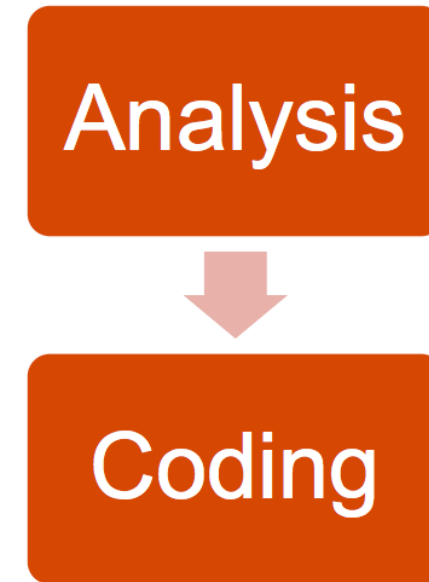


UNIVERSITÀ
DI PARMA

Informatica e Laboratorio di Programmazione
qualità del software
Alberto Ferrari



- *fasi* di sviluppo di una semplice applicazione
 - *analisi*
 - purtroppo spesso trascurata :(
 - *implementazione*
 - codifica in un linguaggio di programmazione

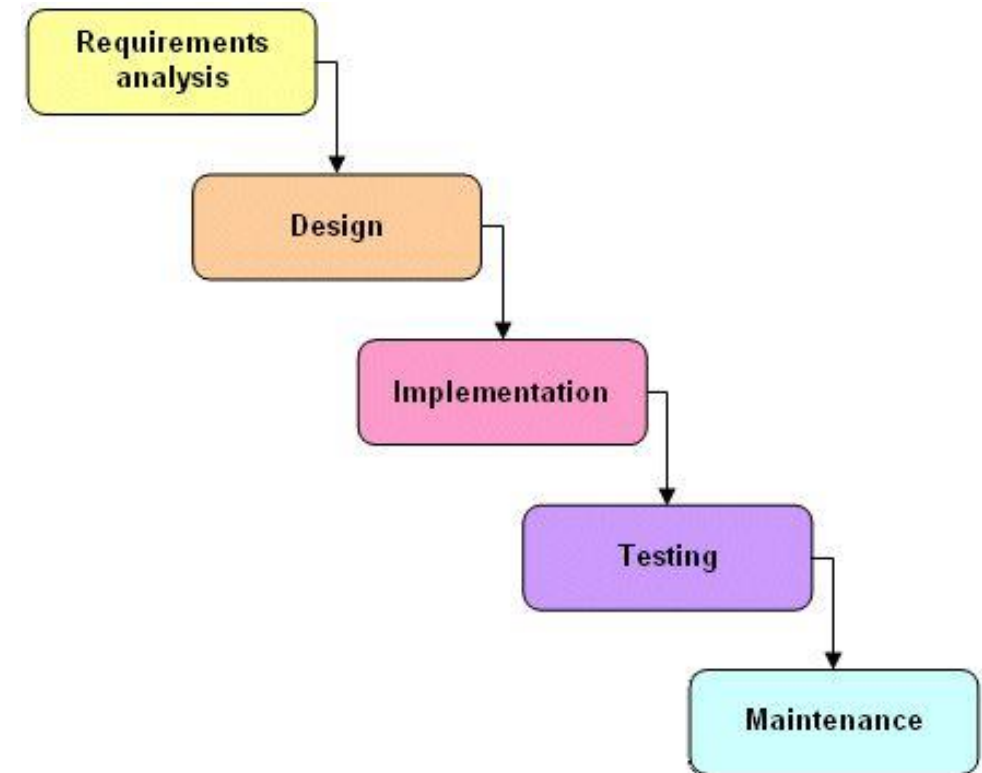


- **software engineering**
- disciplina tecnologica e manageriale che riguarda la *produzione* sistematica e la *manutenzione* dei prodotti software che vengono sviluppati e modificati entro *tempi e costi preventivati*
- nasce negli *anni '60* per l'esigenza di realizzazione di sistemi software di dimensioni e complessità tali da richiedere uno o più team di persone
 - approccio sistematico allo sviluppo, all'operatività, alla manutenzione e al ritiro del software
 - teorie, metodi e strumenti, sia di tipo tecnologico che organizzativo, che consentono di produrre applicazioni con le desiderate *caratteristiche di qualità*

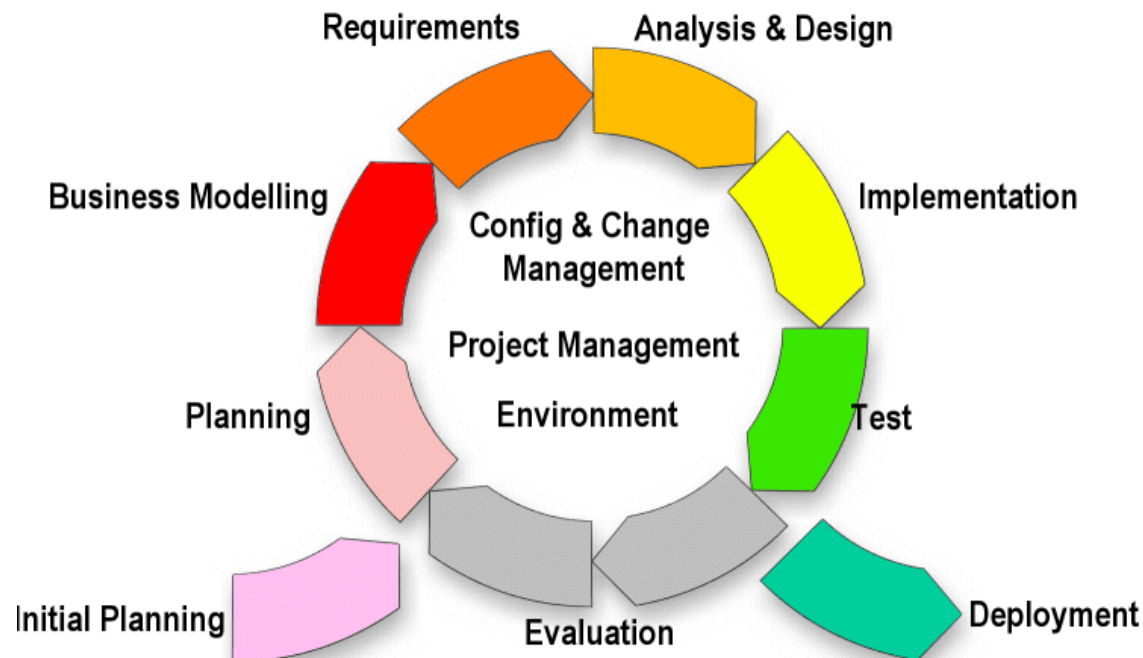
- ***creatività e metodo***
 - il software è un prodotto dell'ingegno e non di un processo industriale
- ***prodotto e processo***
 - l'ingegneria classica (civile, meccanica) progetta il prodotto e anche il processo industriale
 - l'ingegneria del software (spesso) ***progetta solo il prodotto*** e non utilizza un processo industriale formalizzato

- nascita con conferenza NATO del **1968**
 - sw crisis, sw reuse → sw engineering
- anni **1990**, sviluppo delle tecnologie orientate agli oggetti
 - UML (Unified Modeling Language)
 - design pattern
- **metà anni '90**, Java
 - prospettive: Web, e-commerce, interoperabilità
 - linguaggio orientato agli oggetti
 - multi-piattaforma
 - web-oriented
 - ben accettato dalla comunità dell'open source

- **analisi**
 - modello, requisiti, fattibilità
- **progetto e implementazione**
 - componenti architetturali... dettaglio classi
- **collaudo**
 - rispetto requisiti, qualità sw
- **rilascio e manutenzione**
 - 40%-80% del costo totale
 - non noti o non colti correttamente i requisiti
 - cambiano le condizioni operative ...



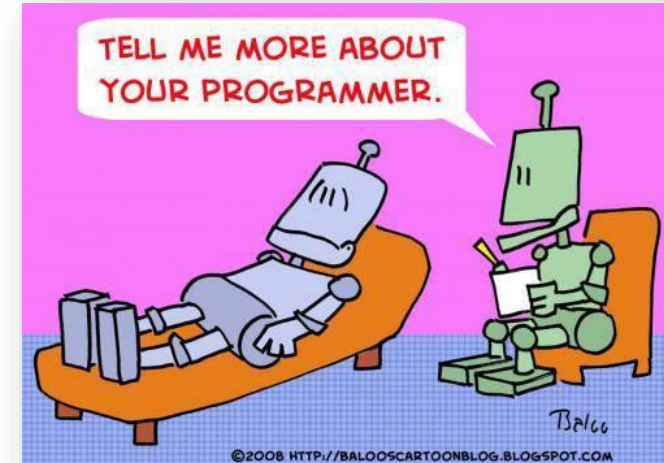
- evoluzione ineliminabile per molti sistemi
 - prestazioni, qualità, funzionalità (manutenzione *perfettiva*, ~60%)
 - anomalie ed errori (manutenzione *correttiva*, ~20%)
 - mutamenti dell'ambiente (manutenzione *adattativa*, ~20%)
- sviluppo iterativo e metodologie agili
 - rilascio frequente ed incrementale
 - <http://agilemanifesto.org/>



- *stiamo scoprendo modi migliori di creare software ... siamo arrivati a considerare importanti:*
 - *gli individui e le interazioni più che i processi e gli strumenti*
 - *il software funzionante più che la documentazione esaustiva*
 - *la collaborazione col cliente più che la negoziazione dei contratti*
 - *rispondere al cambiamento più che seguire un piano*
- *ovvero, fermo restando il valore delle voci a destra, consideriamo più importanti le voci a sinistra*



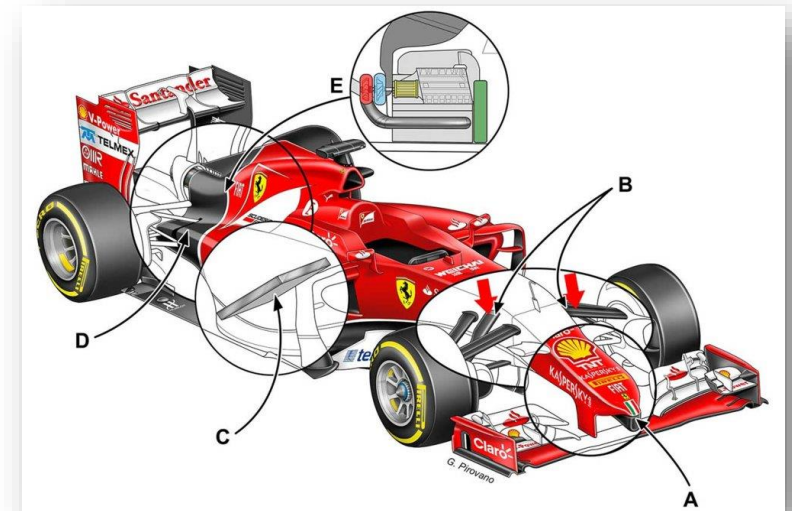
- le qualità su cui si basa la *valutazione* di un sistema software possono essere:
 - *interne*, riguardano le caratteristiche legate al processo di sviluppo e non sono direttamente visibili agli utenti
 - *esterne*, riguardano le funzionalità fornite dal prodotto sw e sono direttamente visibili agli utenti
- le categorie sono legate:
 - *product quality is process quality*



- **correttezza e affidabilità**
 - il sistema rispetta le specifiche, l'utente può affidarsi al programma
- **robustezza**
 - il sistema si comporta in modo ragionevole anche fuori dalle specifiche
- **efficienza**
 - usa bene le risorse di calcolo
- **scalabilità**
 - migliori prestazioni con più risorse
- **sicurezza**
 - riservatezza, autenticazione, autorizzazione, accounting
- **facilità d'uso**
 - interfaccia utente permette di interagire in modo naturale



- **verificabilità**
 - sistema basato su modello formale
- **riusabilità**
 - parti per costruire nuovi sistemi
- **manutenibilità**
 - riparabilità, evolvibilità (nuove specifiche), adattabilità (cambiamenti ambiente)
- **interoperabilità**
 - capacità di co-operare con altri sistemi, anche di altri produttori
- **portabilità**
 - adatto a più piattaforme hw/sw
- **comprensibilità**
 - codice leggibile, documentato



- rispetto a cosa valutiamo *correttezza* e *affidabilità* di un programma?
 - idea del programmatore
 - non formulata, non documentata
 - incompleta, mutevole, facilmente dimenticata
 - *specifiche* (*formali o informali*)
 - formulate, scritte, studiate e condivise
 - parte del progetto e del programma
 - specifiche assiomatiche: espressioni logiche o asserzioni
 - precondizioni, postcondizioni e invarianti

ingegneria del software
design by contract



- ***precondizioni***
 - stabiliscono se ***è possibile*** chiamare un metodo
 - prerequisiti per l'attivazione
- ***postcondizioni***
 - stabiliscono se il metodo restituisce il ***valore atteso***, cioè se produce l'***effetto desiderato***
 - ... in relazione ai parametri (*che soddisfano le precondizioni*)
 - definiscono il ***significato*** del metodo
- divisione delle ***responsabilità*** tra moduli
 - ***errore*** del codice chiamante (***client***) se ***precondizioni*** non soddisfatte
 - ***errore*** del codice chiamato (***server***), se ***postcondizioni*** non soddisfatte

- ***precondizioni + postcondizioni = contratto***
 - ... tra modulo chiamante e modulo chiamato
- infrazione di un contratto: problema serio
 - errore rispetto alle specifiche
 - eccezione e/o terminazione
- ***no*** divisione responsabilità → ***sovrapposizioni***
 - tutti i moduli assumono molte responsabilità
 - programmazione difensiva: tutte le parti del programma controllano tutte le condizioni
 - grosso programma → ancora più grosso

```
def sqrt(x: float) -> float
```

- *precondizioni*: $x \geq 0$
- *postcondizioni*: $\text{abs}(\text{result} * \text{result} - x) \leq 0.00001$
- codice *chiamante*
 - obblighi: deve passare un numero non negativo
 - benefici: riceve la radice del numero
- codice *chiamato*
 - obblighi: restituisce un numero r tale che $r * r \simeq x$
 - benefici: può assumere che x non è negativo

- *espressioni booleane*, simili a predicati matematici
- esprimono *proprietà semantiche* di classi e metodi
- utili per collaudo e debugging, ma anche documentazione
- violazione → **AssertionError**
 - normalmente abort, terminazione programma

```
assert age > 0
```

PYTHON

• Cattura rettangolare

```
assert age > 0;
```

JAVA

```
static_assert(age>0, "Error negative age");
```

C++

- asserzioni utili per:
 - preconditioni, postcondizioni, invarianti di classe
 - invarianti interne e di controllo del flusso
- argomenti di metodi pubblici sbagliati → **eccezione**
 - Python -> *ValueError* o *TypeError*
 - di solito, asserzioni usate per debug...

```
while True:  
    try:  
        val = int(input("eta: "))  
        break  
    except ValueError:  
        print("Attenzione valore inserito non numerico")
```



```
#include <iostream>

using namespace std;

int Fatt(int n) {
    if(n< 0) throw invalid_argument("Negative paramether");
    if (n==0) return 1;
    return n*Fatt(n-1);
}

int main() {
    long fi; int n = -3;
    try {
        fi = Fatt(n); //cause an exception to throw
    }
    catch(invalid_argument& e) {
        cerr << "error: " << e.what() << endl; return -1;
    }
    cout << n << "!=" << fi << endl; return 0;
}
```

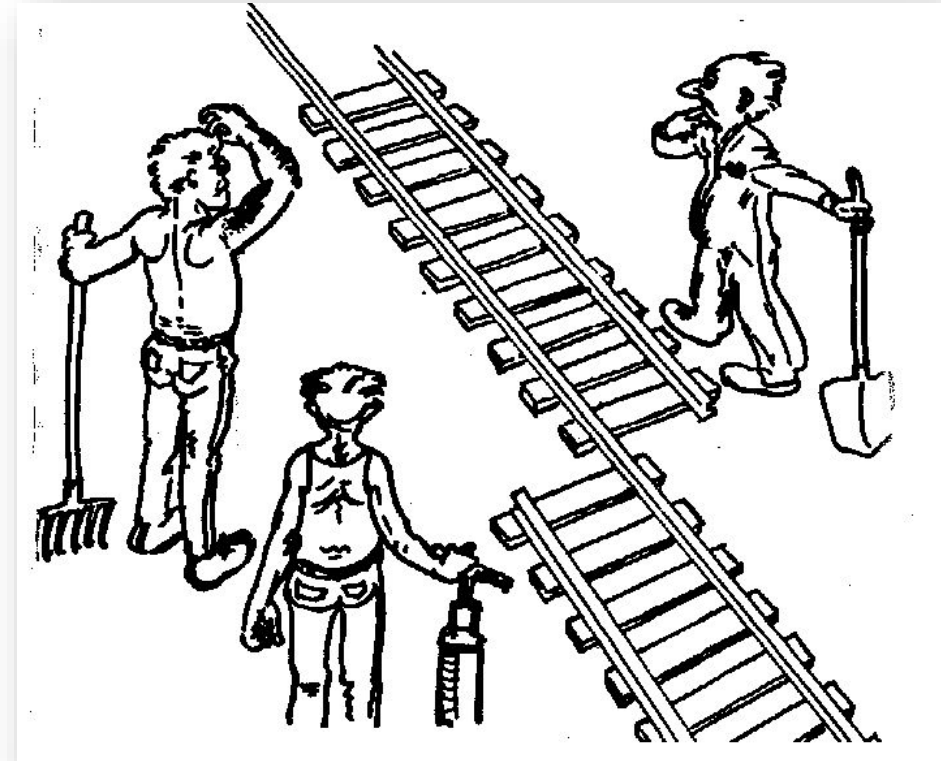
```
from math import sqrt

def my_sqrt(x: float) -> float:
    '''
    Precondition: x >= 0
    Postcondition: abs(result * result - x) <= 0.00001
    '''
    if x < 0: raise ValueError("sqrt: arg < 0")
    result = sqrt(x)
    assert abs(result * result - x) <= 0.00001
    return result

print(my_sqrt(2))
```

```
/**
 * Precondition: x >= 0
 * Postcondition: abs(result * result - x) <= 0.00001
 */
public double sqrt(double x) {
    if (x < 0) throw new IllegalArgumentException("sqrt: arg < 0");
    // ...
    assert Math.abs(result * result - x) <= 0.00001;
    return result;
}
```

verifica e validazione



- obiettivo: mostrare che il sistema ...
 - è **conforme** alle specifiche
 - soddisfa i **bisogni** dell'utente
- la fase comprende
 - revisione
 - collaudo del sistema
- test case, derivati dalle specifiche



- scovare **bug** non è un compito facile, e nemmeno una esperienza eccitante...
 - **costoso**: non è insolito dedicare al testing il 40% del tempo e delle risorse di un progetto
- far emergere bug in prime fasi dello sviluppo!
 - se trovare e correggere un problema in fase di specifica dei requisiti costa **1\$**...
 - **5\$** in progetto, **\$10** in programmazione,
 - **\$20** in unit testing, fino a **\$200** dopo consegna
- alcuni bug possono capitare già a causa di specifiche non ben chiare e capite



- il primo volo dell'*Ariane 5* fallì e il razzo si *autodistrusse* per un malfunzionamento del software di controllo
 - un dato a 64 bit in virgola mobile venne convertito in un
 - intero a 16 bit con segno, questa operazione causò una trap del processore
- per motivi di efficienza i progettisti avevano *disabilitato* il *controllo* software sulle trap
- fu necessario quasi un anno e mezzo per capire quale fosse stato il malfunzionamento e un danno stimato di *500.000.000\$*



- il *Mars Polar Lander* fa parte di una coppia di sonde del Programma Mars Surveyor, assieme al Mars Climate Orbiter
- la missione delle due sonde era di studiare la meteorologia, il clima e le quantità di acqua e di anidride carbonica del pianeta *Marte*, le comunicazioni con il Mars Polar Lander si persero prima del suo ingresso nell'atmosfera marziana
- la teoria più accreditata è che un **errore software** (*una variabile non inizializzata*) abbia fatto spegnere i motori ad una quarantina di metri di quota, per cui la sonda avrebbe impattato il suolo ad un centinaio di km/h



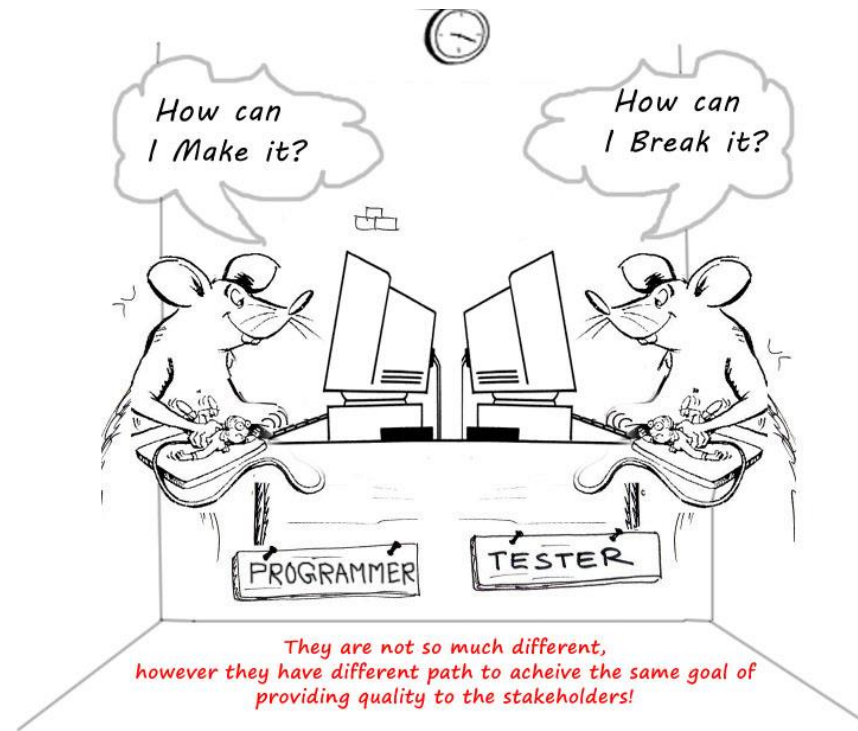
- analisi del codice per capirne le caratteristiche e le funzionalità
- ***code walk-through***
 - selezione porzioni di codice e valori di input
 - simulazione su carta comportamento del sistema
- ***code inspection***
 - uso di variabili non inizializzate
 - loop infiniti
 - letture di porzioni di memoria non allocata
 - rilascio improprio della memoria

- il **test** di un programma prevede l'esecuzione di una o più **serie** di **test-case**
- struttura di un test-case
 - **precondizioni** per la corretta esecuzione del caso
 - **azioni** per l'esecuzione del caso
 - **risultati attesi** dall'esecuzione e i criteri per dichiarare l'esito positivo o negativo

“Le operazioni di testing possono individuare la presenza di errori nel software ma non ne possono dimostrare la correttezza»

(E. Dijkstra)

- il programmatore e il tester hanno *obiettivi contrapposti*
- spesso è preferibile che sia *persone diverse*



- tipi di test
 - *white box* (in the small)
 - *black box* (in the large)
- livelli di test
 - *unit test* (test dei singoli componenti)
 - *integration test* (test globale dopo unit test)
- ripetizione di test
 - *regression test*

- test basati sulla conoscenza della *struttura interna* del codice
- un errore non può essere scoperto se la parte di codice che lo contiene non viene mai eseguita



- sistema = scatola nera
 - si verificano le corrispondenze di input e output
 - white-box testing impossibile per grandi sistemi
 - test case scelti in base alle specifiche dei requisiti
- desiderata: trovare errori...
 - **funzionali**: otteniamo i risultati attesi per i dati input di un metodo?
 - **di interfaccia**: i dati sono passati correttamente tra i metodi?
 - **di efficienza**: il metodo è abbastanza veloce?



- *pianificazione* dei test
- *progettazione* dei test
- *esecuzione* dei test
- *automatizzazione* dei test

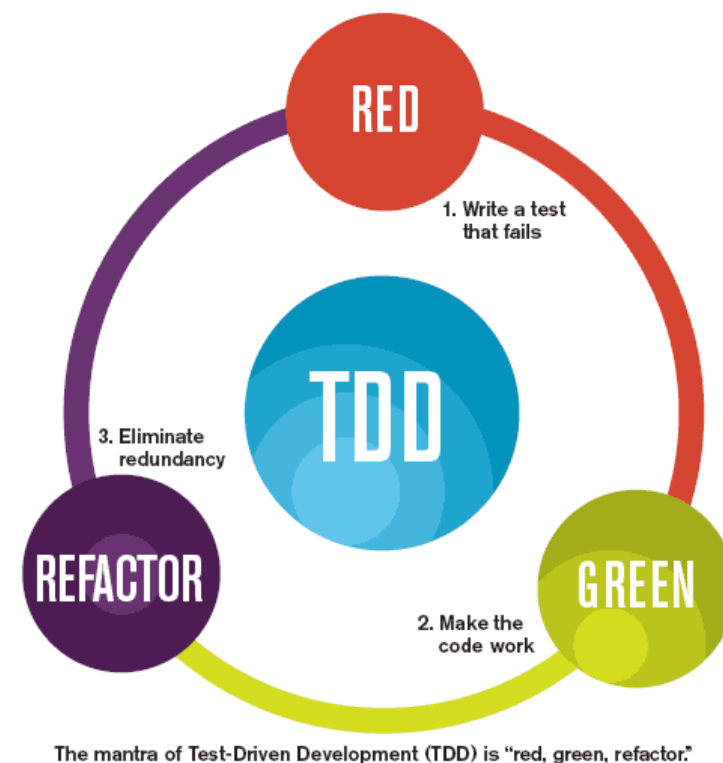


- partizionamento dei dati in ingresso in *classi di equivalenza*
 - irrealistico testare tutti i possibili ingressi (es. sqrt)
 - ipotesi: sufficiente testare un solo caso per classe
 - si includono casi limite e valori non validi
 - precondizioni: riducono il numero di casi di test

```
def swap_elements(v: list, i: int, j: int):  
    '''  
    Exchange element i and j in list v  
    v: empty, one element, more elements  
    i, j: one or both indexes out of range... or both in range: i < j, i > j, i = j  
    '''  
    # ...
```

- scopo: trovare *errori di regressione*
 - errori in un programma che prima era corretto poi è stato *modificato*
 - un errore di regressione è un errore che *prima non c'era*
- dopo la modifica di una parte *P* nel programma *Q*
 - testare che la *parte P* funzioni correttamente
 - testare che l'*intero programma Q* non sia stato danneggiato dalla modifica

- le metodologie agile tendono a enfatizzare il ruolo dei test
- viene teorizzata la codifica dei test prima ancora dello sviluppo del codice dell'applicazione da realizzare
- **Test-Driven Development**
 - nella fase **rossa** si scrive un **test automatico** per la nuova funzione da sviluppare
 - il test fallisce in quanto la funzione non è stata ancora realizzata
 - nella fase **verde** si scrive la **quantità minima** di codice necessaria per passare il test
 - nella fase di **refactoring** si esegue il refactoring del codice per adeguarlo a determinati standard di qualità



- i test ***JUnit*** non richiedono continuo intervento o giudizio da parte dell'utente
- facile eseguire ***molti test assieme***, su un certo progetto
- come definire un test:
 - annotare un metodo di test con `@org.junit.Test`
 - per controllare la validità di una espressione, usare `assertTrue(boolean)`
 - import static: `import org.junit.Assert.*`

```
assertArrayEquals(...) // Asserts that two arrays are equal.  
assertEquals(...) // Asserts that two objects are equal.  
assertFalse(...) // Asserts that a condition is false.  
assertNotNull(...) // Asserts that an object isn't null.  
assertNotSame(...) // Asserts that two objects do not refer to the same object.  
assertNull(...)  
assertSame(...)  
assertTrue(...) // Asserts that a condition is true.  
fail(...) // Fails a test
```

Controllare che una pallina rimbalzi correttamente contro il bordo inferiore

```
@Test
public void testBounceDown() {
    Ball b = new Ball(8, 11); // dx = 1, dy = 1, w = 16, h = 12
    b.move();
    assertTrue(b.getX() == 9 && b.getY() == 10);
}
```


CppUnit is the C++ port of the famous *JUnit* framework for unit testing.

Test output is in XML or text format for automatic testing and GUI based for supervised tests.

If assertion fails, an exception is thrown

MACRO CppUnit ... #include

CPPUNIT_ASSERT(condition):

CPPUNIT_ASSERT_MESSAGE(message,condition):

CPPUNIT_ASSERT_EQUAL(expected,current):

CPPUNIT_ASSERT_EQUAL_MESSAGE(message,expected,current):

CPPUNIT_ASSERT_DOUBLES_EQUAL(expected,current,delta):

- ***unittest*** è il modulo (*battery included*) che fa parte della librerie standard di Python
- permette di eseguire una serie di test su un progetto senza richiedere l'intervento dell'utente
- utilizzo:
 - importare la libreria ***unittest***
 - creare una sottoclasse di ***unittest.TestCase***
 - scrivere metodi di test, denominati con prefisso ***test***
 - per controllare la validità di una espressione, usare ***assertTrue(bool)***, ***assertEqual(a,b)***, ***assertIn(a,b)*** ...

```
import unittest

def fun(x):
    return x + 1

class MyTest(unittest.TestCase):
    def test(self):
        self.assertEqual(fun(3), 4)

unittest.main()
```

<https://docs.python.org/3/library/unittest.html>

Method	Checks that	New in
<code>assertEqual(a, b)</code>	<code>a == b</code>	
<code>assertNotEqual(a, b)</code>	<code>a != b</code>	
<code>assertTrue(x)</code>	<code>bool(x) is True</code>	
<code>assertFalse(x)</code>	<code>bool(x) is False</code>	
<code>assertIs(a, b)</code>	<code>a is b</code>	3.1
<code>assertIsNot(a, b)</code>	<code>a is not b</code>	3.1
<code>assertIsNone(x)</code>	<code>x is None</code>	3.1
<code>assertIsNotNone(x)</code>	<code>x is not None</code>	3.1
<code>assertIn(a, b)</code>	<code>a in b</code>	3.1
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	3.1
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>	3.2
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>	3.2

```
import unittest
from lightsout_console import LightsOut
from random import randrange

class LightsOutTest(unittest.TestCase):

    def test_playat00(self):          # test 0,0
        game = LightsOut(side = 6, level = 0) # 6x6 all False
        game.play_at(0,0)
        for dx, dy in ((0, 0), (1, 0), (0, 1)):
            self.assertTrue(game.value_at(dx,dy) == "@")
        game.play_at(0,0)
        for dx, dy in ((0, 0), (1, 0), (0, 1)):
            self.assertTrue(game.value_at(dx,dy) == "-")
        #self.assertTrue(game.value_at(3,3) == "@")
```

```
    def test_playat_rnd(self):      # test random
        side = randrange(10)
        game = LightsOut(side, level = 0) # sidexside all False
        x = randrange(side)
        y = randrange(side)
        game.play_at(x,y)
        for dx, dy in ((0, 0), (0, -1), (1, 0), (0, 1), (-1, 0)):
            if 0 <= x+dx < side and 0 <= y +dy < side:
                self.assertTrue(game.value_at(x+dx,y+dy) == "@")
        game.play_at(x,y)
        for dx, dy in ((0, 0), (0, -1), (1, 0), (0, 1), (-1, 0)):
            if 0 <= x+dx < side and 0 <= y +dy < side:
                self.assertTrue(game.value_at(x+dx,y+dy) == "-")

if __name__ == '__main__':
    unittest.main()
```

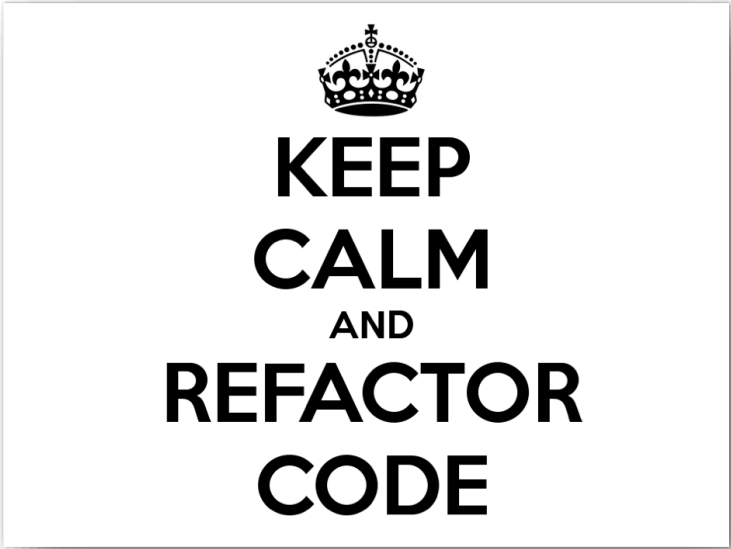
- test ripetuto con diversi parametri
- prima soluzione:
 - un test case per ogni gruppo di parametri
 - per alcune applicazioni implica una enorme quantità di test
- seconda soluzione (semplicistica):
 - test contenente un ciclo
 - ad ogni iterazione utilizza un gruppo di parametri diversi
 - vengono eseguite le istruzioni da testare sui nuovi parametri
 - problema: il test si blocca al primo errore
- terza soluzione (sottotest):
 - utilizzo subTest (<https://docs.python.org/3/library/unittest.html>)

- oltre a controllare se il codice che completa correttamente l'esecuzione nei *casi normali*...
- è necessario anche verificare se, in situazioni eccezionali, il codice mostra il comportamento atteso
- per verificare se una eccezione attesa sia effettivamente sollevata:
 - usare il metodo *assertRaises*

```
def test_out_of_arena(self):  
    with self.assertRaises(ValueError):  
        b = Ball(-1, -1)
```

- **fixture**: permette di gestire operazioni comuni a più metodi
- se ci sono diversi test con una fixture comune...
 - aggiungere alla classe campi per le varie parti della fixture
 - inizializzare questi campi, nel metodo **setUp**
 - liberare eventuali risorse allocate, nel metodo **tearDown**
- Una volta creata la fixture, può essere usata da tutti i test case
- setUp e tearDown vengono eseguiti prima e dopo ogni test della classe

<https://docs.python.org/3/library/unittest.html>



**KEEP
CALM
AND
REFACTOR
CODE**

code refactoring

- il *code refactoring* è una tecnica per modificare la struttura interna di porzioni di codice senza modificarne il comportamento esterno
- l'obiettivo è *migliorare* alcune *caratteristiche non funzionali* del software:
 - *leggibilità*
 - *manutenibilità*
 - *riusabilità*
 - *estendibilità*
 - eliminazione di *code smell*
- molti ambienti di sviluppo offrono valide funzionalità di ausilio al refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior

[Martin Fowler]

... anche per software di piccole dimensioni

consigli pratici

- evitare *code smell*:
 - identificatori non significativi (i,j,k ...)
 - valori "cablati" nel codice (hard code)
 - copy and paste programming (Don't Repeat Yourself!)
- *anti-pattern*
 - *dead code* (codice irraggiungibile)
 - *spaghetti code* (flusso incomprensibile)
 - kitchen sink (lavello della cucina) o swiss army knife ("coltellino svizzero")
 - classe che contiene un gran numero di operazioni complesse ed eterogenee tra loro



- *indentation*
- *breaking long lines and strings*
- *placing Braces and Spaces*
- ...
- *functions*
- *commenting*

“This is a short document describing the preferred coding style for the linux kernel. Coding style is very personal, and I won't force my views on anybody, but this is what goes for anything that I have to be able to maintain, and I'd prefer it for most other things too. Please at least consider the points made here. First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture [Linus Torvalds]”

[Linux Kernel Coding Style](#)

- usare un ***debugger*** per valutare espressioni in fase di esecuzione
 - si può decidere cosa valutare a seconda del flusso di esecuzione e dei valori generati, senza ricompilare
- istruzioni di ***stampa*** all'interno del programma
 - valore di espressioni scritto a ***console*** o su file di log
- entrambi gli stili, ***scarsamente automatizzati***
 - necessità di intervento attivo durante l'esecuzione dei test
 - giudizio dei risultati da parte dell'utente
 - quali valori analizzare? sono coerenti?
- ***scarsamente componibili***
 - difficile controllare molte espressioni nel debugger
 - «***scroll blindness***»: troppe istruzioni di stampa ⇒ *codice poco leggibile*