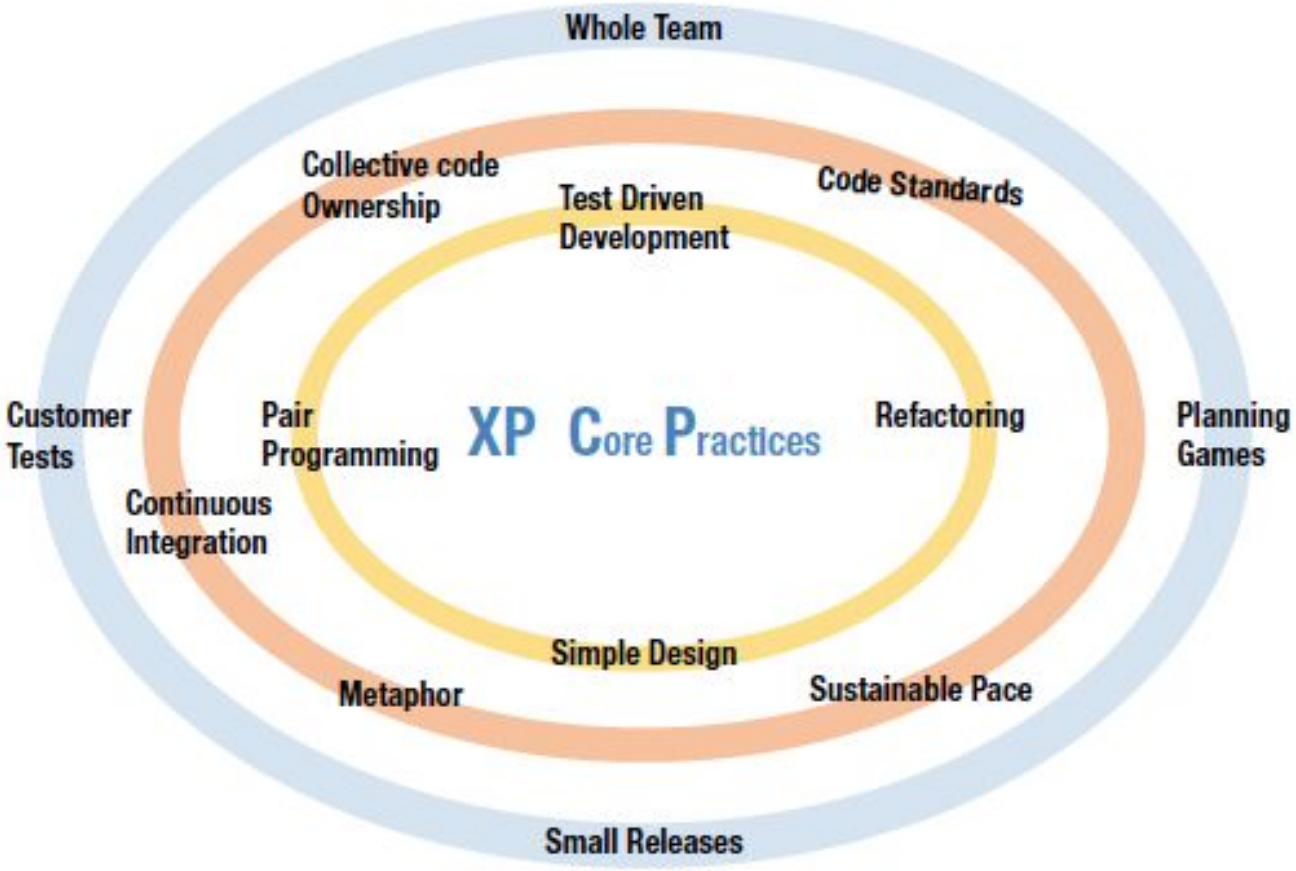
Testing
1- TFD-TDD-PP

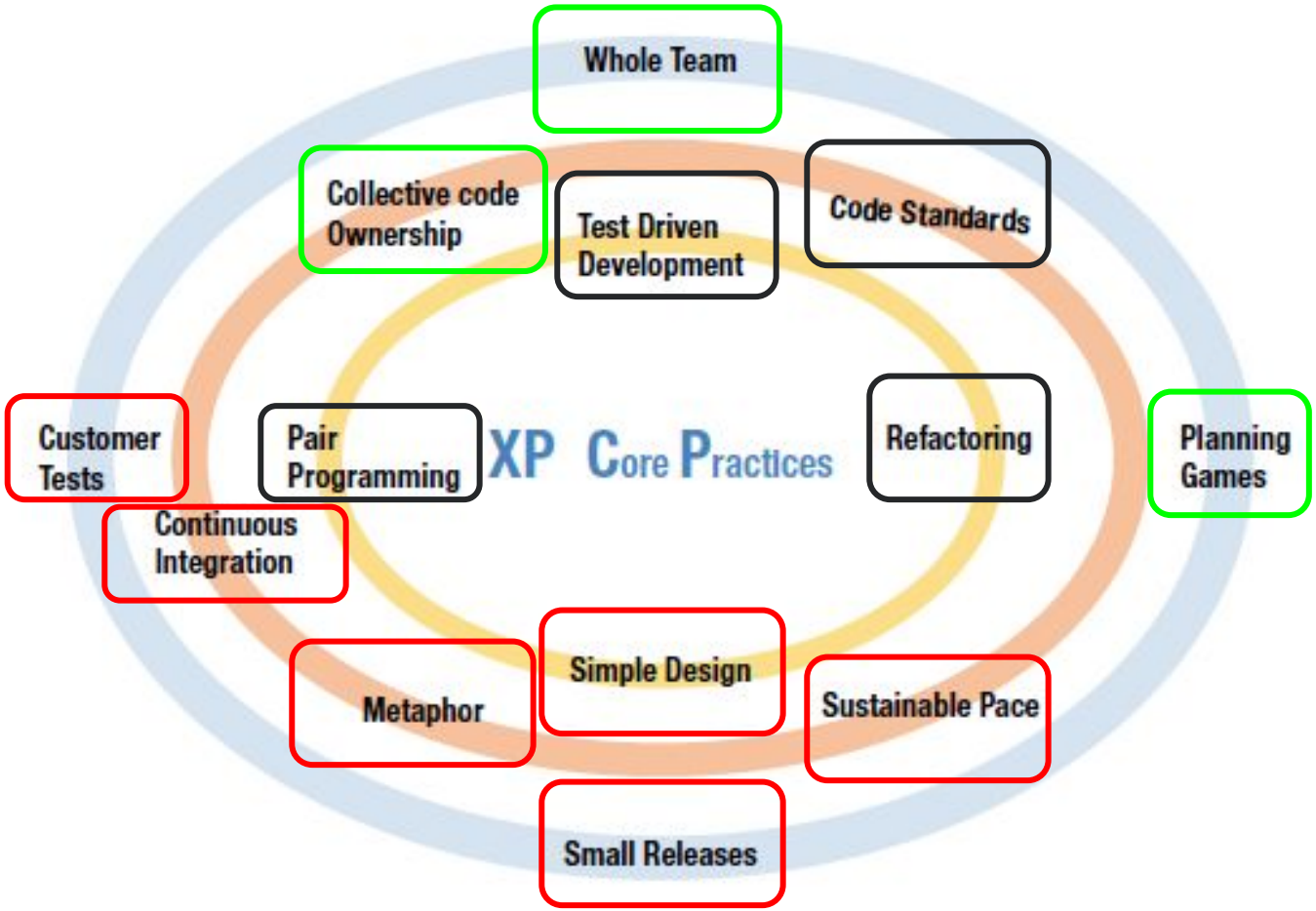
Prof. Marcello Missiroli (basato anche sul lavoro di Marco Fracassi & Roberto Grandi (7pixels)

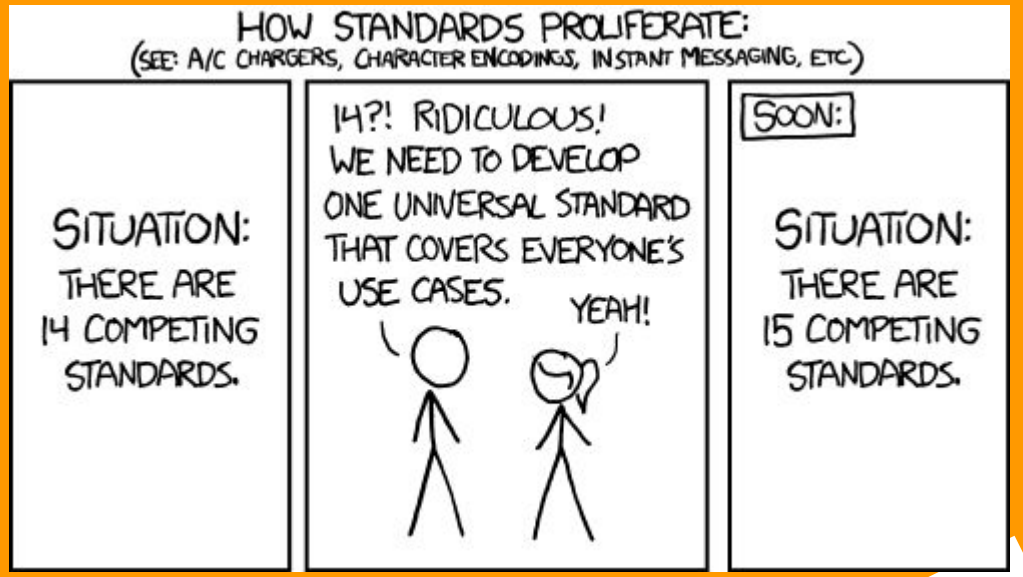
1.

Recap

Le pratiche agili


Whole Team
Collective code Ownership
Test Driven Development
Code Standards
Customer Tests
Pair Programming
XP Core Practices
Refactoring
Planning Games
Continuous Integration
Simple Design
Metaphor
Sustainable Pace
Small Releases


Whole Team
Collective code Ownership
Test Driven Development
Code Standards
Customer Tests
Pair Programming
XP Core Practices
Refactoring
Planning Games
Continuous Integration
Metaphor
Simple Design
Sustainable Pace
Small Releases



2.

Coding conventions

Coding standards

- Sono approvati dal Team di Sviluppo, non imposti. Supportano anche la pratica CoCoOw.
- Appoggiarsi a standard esistenti (integrazione IDE)
- Esempi:
  - Uso di Javadoc (o analogo)
  - Usare standard affermati: Google, Oracle (Java), PEP (Python), Google, CKAN (Javascript)

Coding standards personali

- Ogni team PUO' darsi regole specifiche,
- Esempi:
  - Uso del formato di variabili polacco (usato da Microsoft ad esempio): int64Hello, szInput..
  - Nessun metodo deve avere più di 16 righe
  - Non usare mai metodi statici (applicare Singleton)

Troppo personale?

Silicon Valley

Stagione 3 ep.6

2.

Testing

DI CHE SI PARLA?

Sappiamo (almeno in teoria) come funziona il debugging. E preferiamo evitarlo: una pratica lunga, noiosa, e non con risultati certi.

Come possiamo assicurare (o quantomeno migliorare) la qualità del nostro codice, specie in un contesto di “programming in the large”?

Semplice: facendo fare il lavoro sporco al computer.

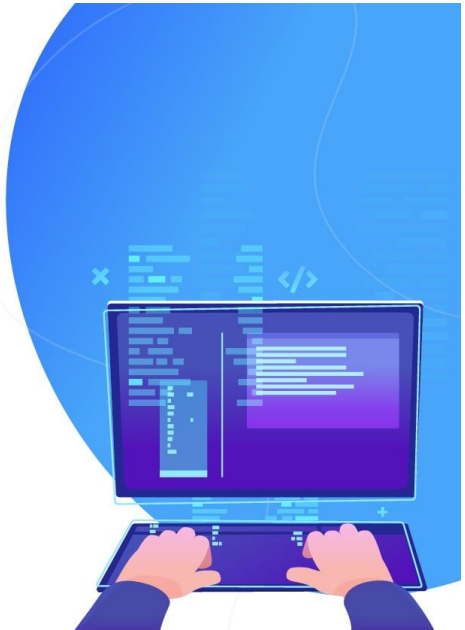
Vi presentiamo i test automatici.

Progresso significa anche questo...

Testing

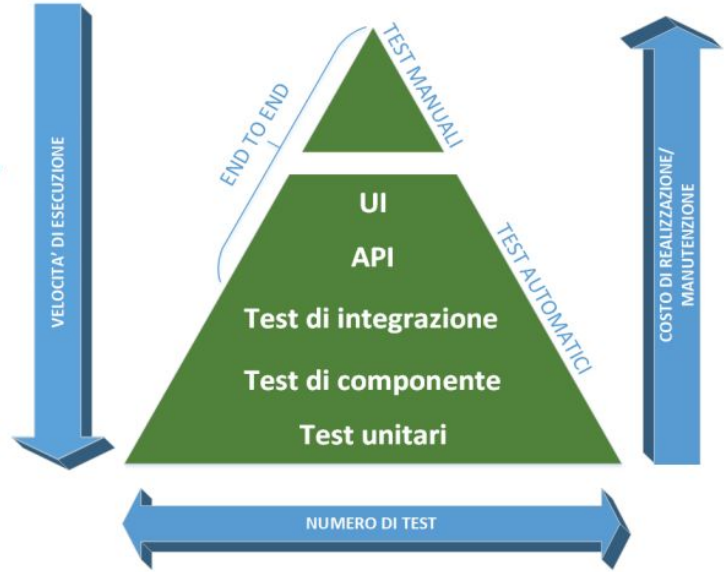
- I test sono l'UNICO modo che il programmatore ha per verifica se il codice scritto funziona oppure no.
- Tipicamente, il programmatore ingenuo lancia il programma 1-2 volte e, se tutto va bene, si considera soddisfatto.
- Questo però non è quasi mai sufficiente: occorre un approccio più metodico.

> ALL CODE IS GUILTY UNTIL PROVEN INNOCENT



Tipologia di test

- Il 70% dei test dovrebbero essere test di unità
- I test manuali dovrebbero essere ridotti all'osso.
- Automatizzare il più possibile permette di risparmiare un sacco di tempo.
-



Unit testing: definizione

Per **Unit testing** si intende l'attività di testing (prova, collaudo) di singole unità software. Per unità software si intende normalmente il minimo componente di un programma dotato di funzionamento autonomo come

- una singola funzione
- una singola classe
- singolo metodo nella programmazione a oggetti.

Da Wikipedia

Unit testing: caratteristiche

- Automatizzato e ripetibile
- Facile da implementare
- Facile da eseguire (Run → Run Test)
- Esecuzione rapida
- Il codice di entra a far parte del progetto a pieno titolo

2.

Come scrivere test

Test loop

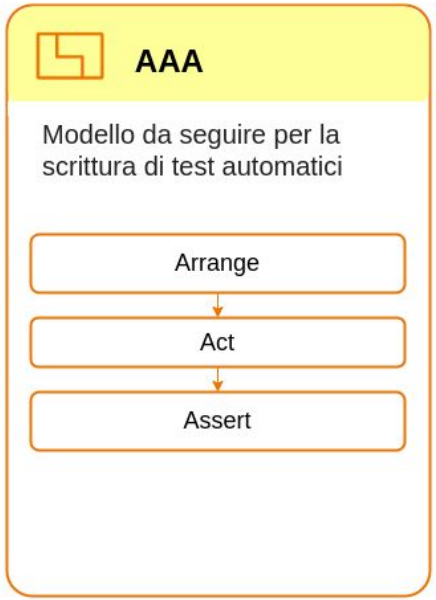
do {

- Scegliere l'obiettivo: cosa testare?
- Scrivere un codice efficace per il controllo
- Eseguire il codice
- Se il codice non passa, debuggare

} while (non si trovano più bug);

Metodologia di scrittura test unit

- **Arrange (organizzare).** che cosa occorre? che dati di input servono?
- **Act (agire).** richiamare una funzione o un metodo e ottenere un risultato.
- **Assert (affermare).** verificare che i risultati coincidano con ciò che ci aspettavamo.



Quando smettere?

```
while (non si trovano più bug);
```

In realtà questa condizione non è facile verificare. In genere si lavora finché si ha tempo e si è “ragionevolmente sicuri” di non trovare nuovi bug. Esiste tuttavia un indicatore “oggettivo”, chiamato Code Coverage, che ci dà una misura di quanto siamo/siete stati bravi a testare il codice.

TESTING IS AN INFINITE PROCESS OF COMPARING THE INVISIBLE TO THE AMBIGUOUS IN ORDER TO AVOID THE UNTHINKABLE HAPPENING TO THE ANONYMOUS.

3.

Test automation


E2E Tests
Integration Tests
Unit Tests
JUnit 5
behat
Supertest
PHPUnit
JUnit 5
Jest
phpspec
nunit
PHPUnit

Tutte le strade portano a Roma

La scelta del metodo da utilizzare per utilizzare i test automatici dipende principalmente dal linguaggio utilizzato, e - secondariamente - dall'IDE utilizzato..

Java

La soluzione decisamente più utilizzata è la libreria Junit, considerata lo standard e giunta alla sua quinta release, denominata Jupiter. La libreria è stata talmente importante che è disponibile per diversi altri linguaggi, complessivamente noti con il nome di xUnit (Cppunit, PhpUnit, ...) ed ha comunque influenzato la scrittura di tutti gli altri sistemi di test.

Java + IDE

Tutti gli IDE permettono di utilizzare Junit, spesso senza aggiunte di software particolari.Eclipse, Netbeans, IDEA, ad esempio, sono pre-configurati per lavorare con JUnit quasi “out-of-the-box”.

Python

Situazione è molto simile alla precedente: la libreria ***unittest*** è di fatto considerata lo standard, ed è largamente utilizzata dai professionisti - anche se esistono librerie alternative. Per gli IDE vale il discorso precedente

Javascript

In Javascript invece non esiste un unico vincitore: ne esistono almeno 5, ciascuno con punti di forza e debolezza. Richiedono tutti un minimo di configurazione, e l'integrazione con l'IDE non va sempre liscia. Citiamo:

- Jest
- Puppetter
- Mocha
- Jasmine
- QUnit

C/C++

Il caso di C/C++ è decisamente più complicato, anche per il fatto che si tratta di un linguaggio compilato. Le librerie sono moltissime, come l'immancabile **Cppunit**, il pervasivo **Googletest**, il completo **Boost.Test** (parte della libreria Boost) e il minimalista **Catch2** .

C#

MSTest è un framework offerto da Microsoft, ottimo per parire

Nunit nasce come porting di Junit, molto veloce

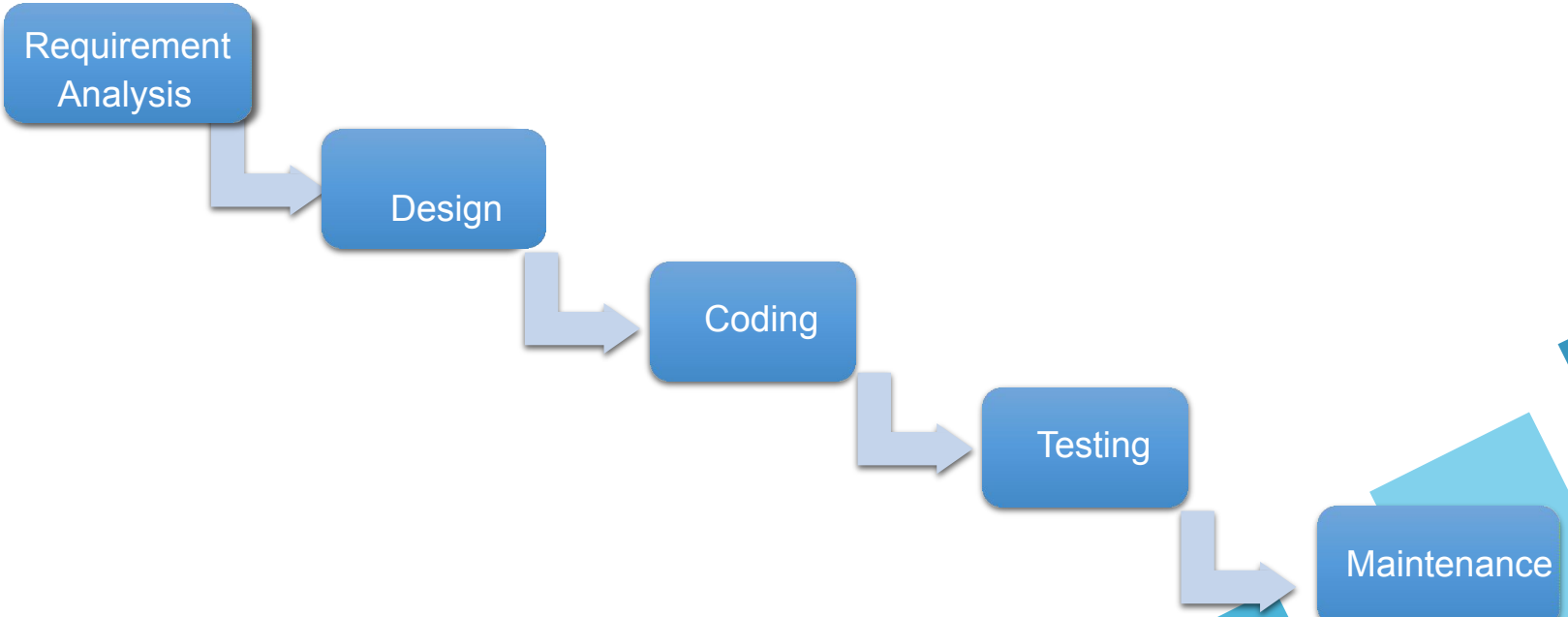
Xunit.net framework open source, estensione ideale di Junit. Forse il più completo.

4.

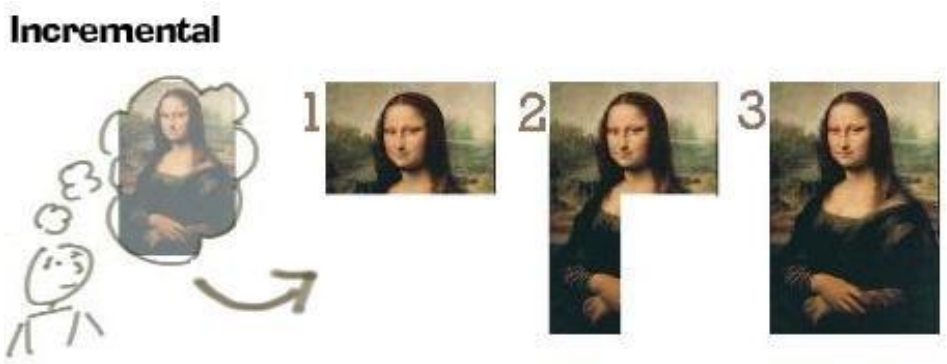
TDD - What & Why?

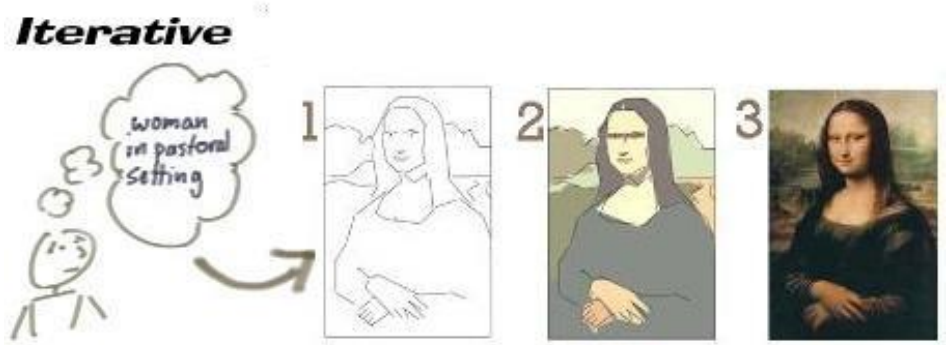
Reversed perspective

SW Engineering – Classical WaterFall Model



SW Engineering - Incrementale/Iterativo





Domande chiave:

- WHAT is TDD
- HOW is it performed?
- WHY do we do it?

Perché fare TDD

- Brainstormate tra voi
- 90 secondi
- Ipotizzate ipotizzate almeno un vantaggio e uno svantaggio.

Perché fare TDD (possibili risposte)

- Design evolutivo del codice
- Focus su un unico aspetto utile
- Sicurezza sul codice di produzione
- Documentazione sempre aggiornata
- Esprimere creatività
- Dare un ritmo allo sviluppo

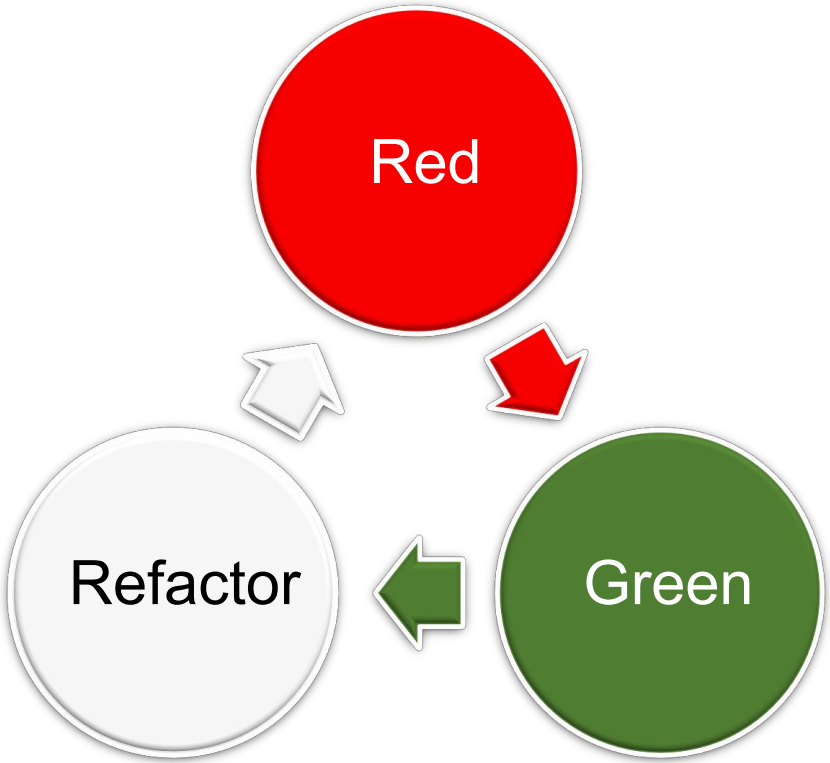
Possibili problemi del TDD

- Più codice da scrivere e manutenere
- Difficile da applicare ad integration testing
- Richiede cambio di mentalità
- Difficile da applicare al codice legacy
- Refactoring spesso posto in secondo piano.

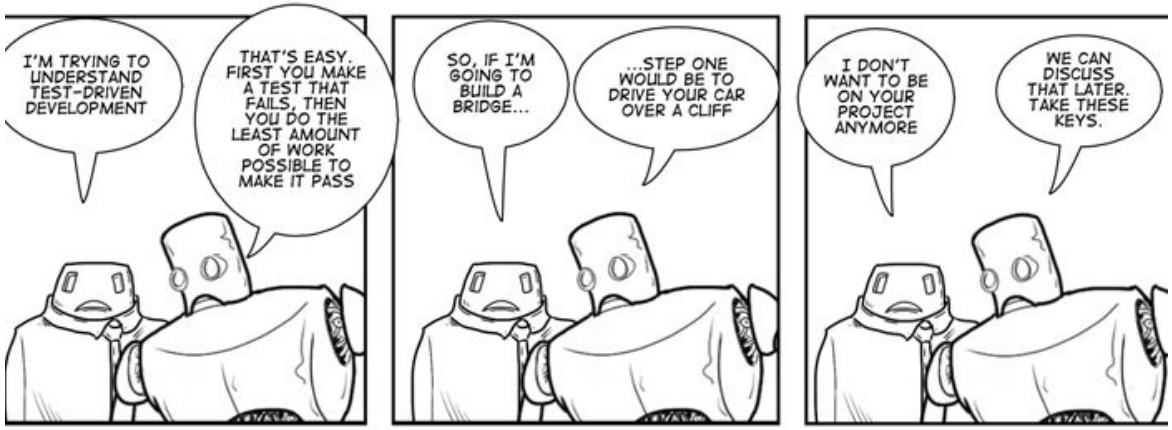
5.

TDD - How?

Come si fa TDD



Red	Scrivo un test che fallisce
Green	Scrivo il codice che fa passare il test
Refactor	Pulisco e il codice ed il test

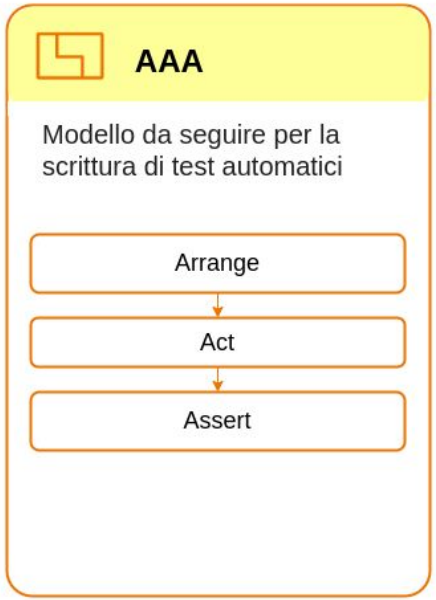

I'M TRYING TO UNDERSTAND TEST-DRIVEN DEVELOPMENT
THAT'S EASY. FIRST YOU MAKE A TEST THAT FAILS, THEN YOU DO THE LEAST AMOUNT OF WORK POSSIBLE TO MAKE IT PASS
SO, IF I'M GOING TO BUILD A BRIDGE...
...STEP ONE WOULD BE TO DRIVE YOUR CAR OVER A CLIFF
I DON'T WANT TO BE ON YOUR PROJECT ANYMORE
WE CAN DISCUSS THAT LATER. TAKE THESE KEYS.

Le 3 regole del TDD

Se voglio fare TDD devo rispettare le seguenti regole

- Non scrivere codice di produzione prima di aver scritto un test rosso
- Non scrivere più di un test che fallisce, un errore di compilazione è come un fallimento
- Non scrivere più codice di produzione di quello necessario per far passare il test correntemente rosso

Le tre A dei test



Come scrivere i test (esempio)

Voglio testare le funzionalità base di una classe conto bancario (Java + JUnit)

Arrange →
```
BankAccount account = new BankAccount().WithBalance(10);
```

Act →
```
account.addMoney(10);
```

Assert →
```
assertEquals(20, account.getBalance());
```

Con quale test partire?

- Happy Path: il caso più semplice/favorevole che porta maggior valore
- Sad Path: casi con input errato.
- ..
- Boundaries Condition: i casi limite

Come scrivo i test - Esempio Boundaries

Voglio testare le funzionalità base di una classe conto bancario (Java + JUnit)

```
BankAccount account = new
BankAccount().WithBalance(0);

account.addMoney(0);

assertThat("Balance must be 0",
account.getBalance(), is(0));

account.addMoney(null);

assertThrown(account.getBalance());
```

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

Ovvero: l'esecuzione di azioni di ristrutturazione del codice volte a migliorarne la struttura interna, abbassandone la complessità senza modificarne le funzionalità

Cos'è il refactoring

- Si basa su una serie di piccole trasformazioni che preservano il comportamento; ogni trasformazione (refactoring) è limitata, ma una sequenza di queste può portare ad una significativa ristrutturazione.
- Poichè ogni trasformazione è limitata è poco verosimile abbia un impatto negativo ed elevato: è controllabile.
- Dopo ogni refactoring, grazie ai test soprattutto, il software mantiene inalterate le funzionalità per definizione (se abbiamo fatto un refactoring)
- Si riduce, grazie ai test, la possibilità di introdurre errori nel software durante le ristrutturazioni del codice

Benefici del Refactoring

- Non far degradare il design del SW
- Eliminare codice duplicato (!!!)
- Rendere il codice più facile da comprendere (per il team nonché per i prof)
- Trovare Bug (se un test fallisce sono già a metà dell'opera)
- Velocizzare lo sviluppo

Code Smell

- Da dove parto a fare refactoring? Dai Code Smell!
-
- "Insieme di caratteristiche che il codice sorgente può avere e che sono generalmente riconosciute come probabili indicazioni di un difetto di programmazione"
- Non sono bug! (il software funziona correttamente)
- Sono "debolezze" del codice che ne riducono la qualità

Il sito “definitivo” di riferimento:
https://refactoring.guru/refactoring/smells

Simple design

- Tutti i test sono verdi
- Non c'è duplicazione (Codice duplicato?, logica duplicata? Algoritmi interscambiabili?)
- L'intento del programmatore è espresso chiaramente
- Il numero di classi e metodi è minimo

Come dovrebbe essere un test?

- Veloce (Li dobbiamo lanciare centinaia di volte! Domanda: Tutti?)
- Ripetibile
- Indipendente (dagli altri test, altrimenti si diventa pazzi!)
- Isolato (dal codice di produzione)
  - Separazione progetti A
  - Anche se molti tool non partono così (test e src vicini di casa)!
  - A volte promuoviamo classi di test a classi applicative

Come dovrebbe essere un test? (2)

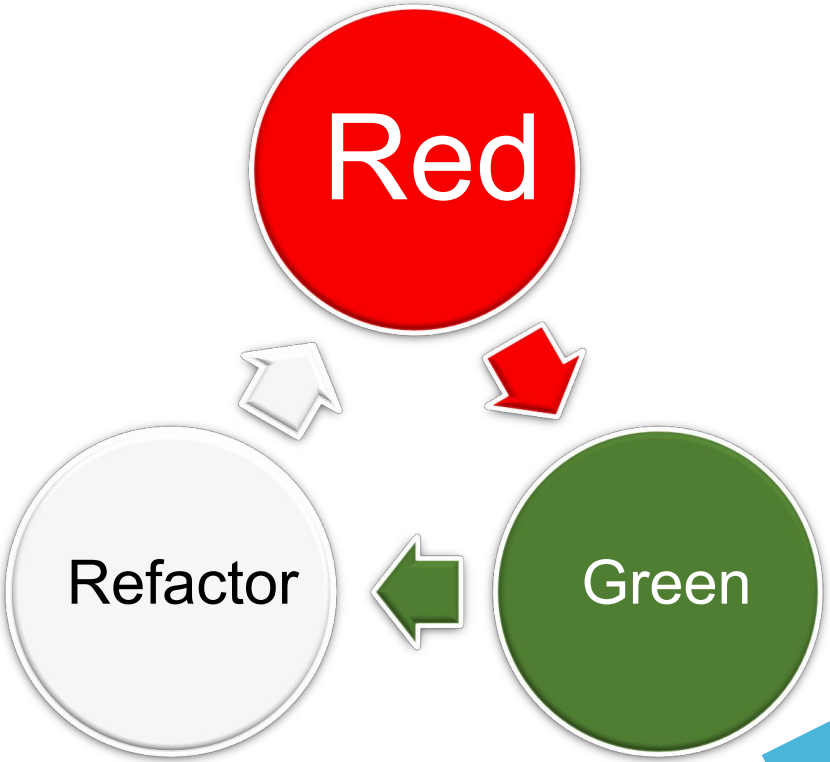
- Rifattorizzato (testare le primitive di test)
- Non distruttivo
  - Preserva le condizioni iniziali (No Drop table)
- Self verified
  - E’ chiaro perché fallisce? Se fallisce devo capire al volo come mai
  - Piuttosto metto un messaggio nelle asserzioni
- Timely (eseguito in tempo ragionevole)

Chi testa il test?

60 secondi per pensarci a coppie....

Il QA?
Il PO?
Lo SM?
Il mio compagno di pair?
Un tester del test?
Un test automatico del test?

Test del Test:
RED verifica il test!



6.

Pair programming

Rythm & Flow

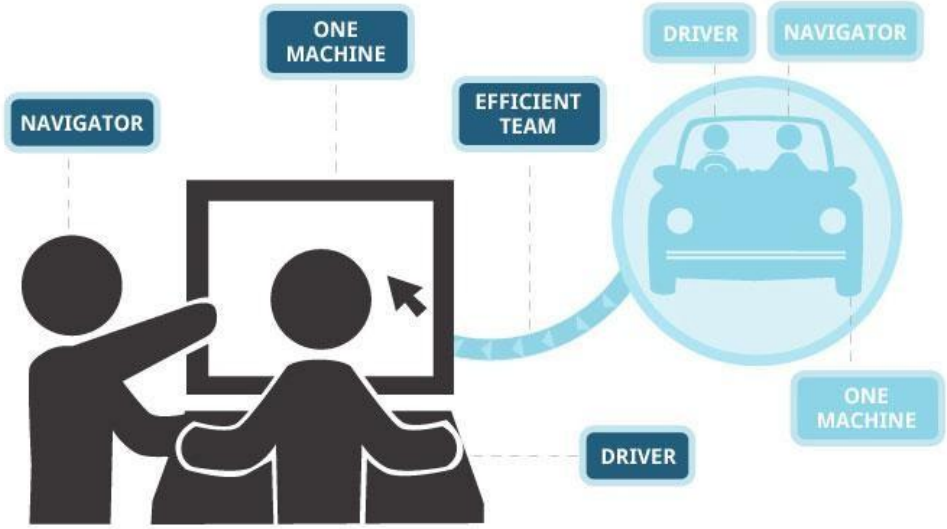
Il ritmo dello sviluppo

- Come possiamo introdurre il concetto di ritmo nella giornata dello sviluppatore?
- Possiamo usare uno strumento che ci permetta anche di valutare il tempo speso?
- Quale unità di misura scegliere?

Proposta

```
{
“name”: “tecnica del pomdoro”,
“youtube”:
https://www.youtube.com/watch?v=CT
70iCaG0Gs,  “website”:
“http://pomodorotechnique.com/”
}
```

Pair Programming



Pair Programming

I programmatori lavorano sempre in coppia, alternandosi tra i ruoli di Driver e Navigator.

DRIVER

- Ha mouse e tastiera
- Scrive i test ed il codice
- Ha una visione di breve periodo

NAVIGATOR

- Ha una todo-list
- Appunta i test mancanti
- Appunta refactoring possibili
- Ha una visione di lungo periodo

Stili e varianti

1. **Classica**: Driver scrive codice, Navigator commenta controlla ad alto livello
2. **Hard**: Navigator detta le idee, Driver esegue - *“For an idea to go from your head into the computer it MUST go through someone else's hands”*
3. **Tour** : Driver fa tutte le scelte raccontando quel che fa, Navigator segue il ragionamento e controlla
4. **Mob** : un intero gruppo partecipa come Navigator, un solo driver

Pair Programming: annedottica

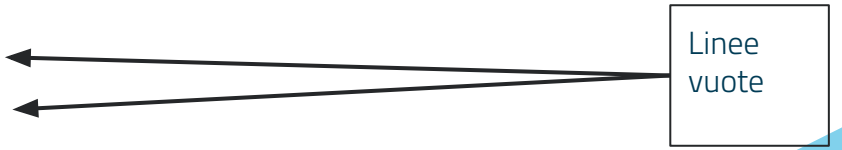
	Progetto1: solisti	Progetto2: coppie
Dimensione (KLOC)	20	520
Team	4	12
Sforzo (mesi persona)	4	72
Produttività (KLOC/mp)	5	7.2
Produttività (KLOC/mpair)	n.d.	14.4
Difetti test unità	107 (5.34 difetti/KLOC)	183 (0.4 difetti/KLOC)
Difetti test integrazione	46 (2.3 difetti/KLOC)	82 (0.2 difetti/KLOC)

Pair Programming - Accounting

Git: remote condiviso

.

```
$ git commit -m "Esempio di
commit.
>
>
Co-authored-by: piffy
<piffy@gmail.com>
Co-authored-by:
prof.missiroli
<prof.missiroli@unimore.it>"
```



Tag introdotto da Github, presto acquisito da tutti

Buone norme di convivenza

- Non interrompere
- Suggerire («cosa ne pensi?»)
- Appuntarsi i problemi emersi – checklist
- Igiene personale

The dark side of pair programming.

Pair Programming - Note

A) Non è detto che le coppie siano fisse. Anzi, in molti contesti si obbliga una turnazione stretta in modo da amalgamare gli stili programmazione ed eliminare la “Silo culture”.
B) Anche il miglior pilota ha bisogno di riposo e anche il miglior navigatore vuole adrenalina. Morale: cambiare ogni 2-3 pomi, seguire user story diverse, affrontare le cose che si conoscono meno

7.

TDD Calisthenics

Toolz

- Linguaggio di programmazione: Java
- Un sistema di versionamento: Git
- Un ambiente di sviluppo: come IDE di riferimento utilizzeremo IntelliJ IDEA, senza supporto di sistemi di building come Maven, Gradle e altro.
- Un Test Framework: Ci concentreremo su Junit 5 (Jupiter). Questa versione permette anche di eseguire test scritti nelle versioni precedenti (JUnit Vintage).

Goal

Vogliamo costruire una calcolatrice in grado di sommare i numeri presenti una stringa di testo

- La stringa può avere più numeri separati dalla virgola
- Se non ho numeri voglio ottenere 0
- ….ho dimenticato qualcosa?

LINK: https://osherove.com/tdd-kata-1

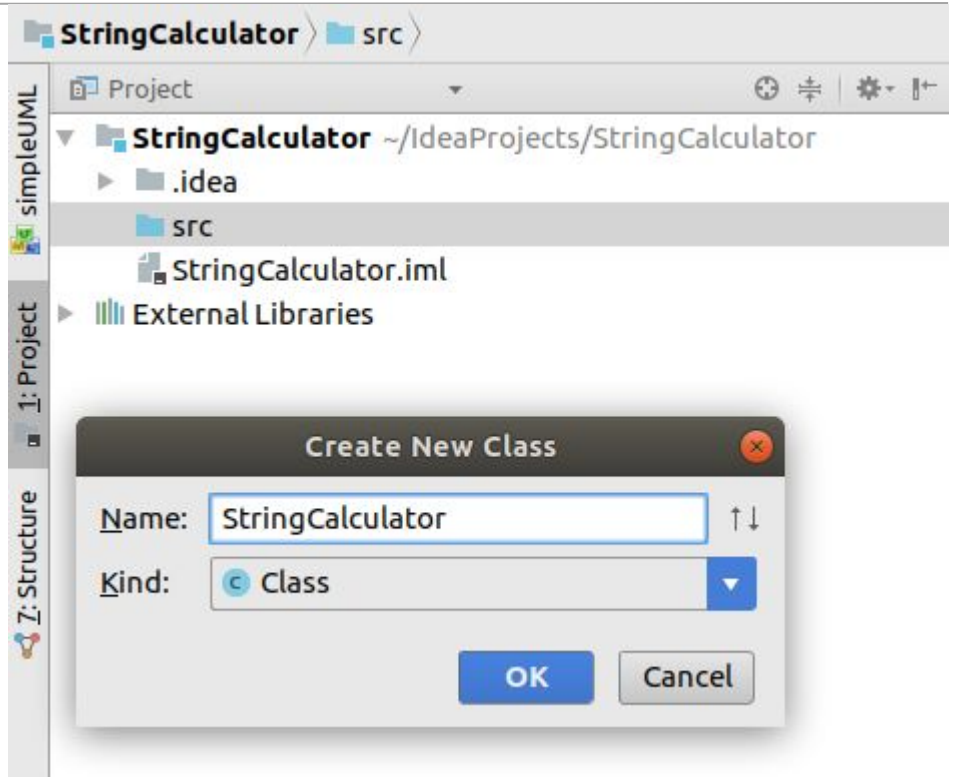
Iniziamo a lavorare

METTETEVI A COPPIE STABILITE DRIVER & NAVIGATOR

Step 1a - Project Setup

IntelliJ

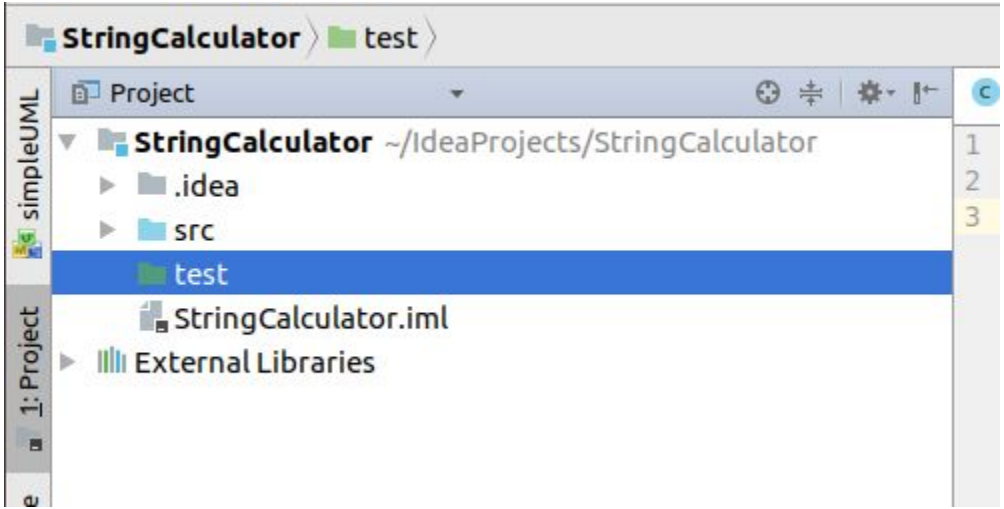
- *Creare un normale progetto Java.*
- *Create la classe StringCalculator*



Step 1b - Project Setup

IntelliJ

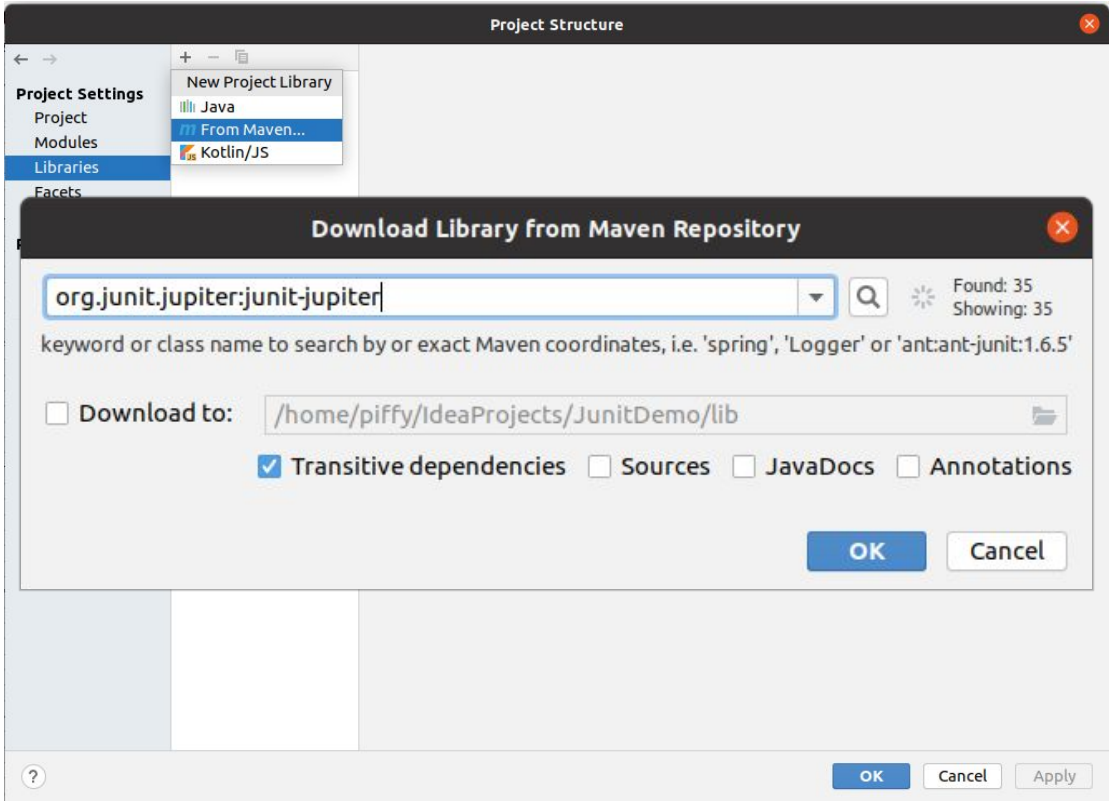
- *Create una cartella chiamata test*
- *Click Destro, Mark directory as > Test source root*
- *Dovrebbe diventare verdino*



Step 1c - Project Setup

IntelliJ

- *Fare ALT-invio sul nome della classe e scegliere “Create test”*
- *Scrivere org.junit.jupiter:junit-jupiter e premere ricerca. Selezionare quindi una versione dal drop-down menu (tipo 5.8.1)*



Step 1d - Project Setup

IntelliJ

Right-click sul test e fate “Run StringCalculatorTest” (oppure create una configurazione).

Struttura base di un test case JUNIT

```
@Test
    public void nomeDelTest() {

      -> SETUP DELLE VARIABILI (A…)
      -> INVOCAZIONE DEL METODO/FUNZIONE (A…)
      ->  UNA O PIU' ISTRUZIONI DI CONTROLLO (A…)

    }
```

Un test

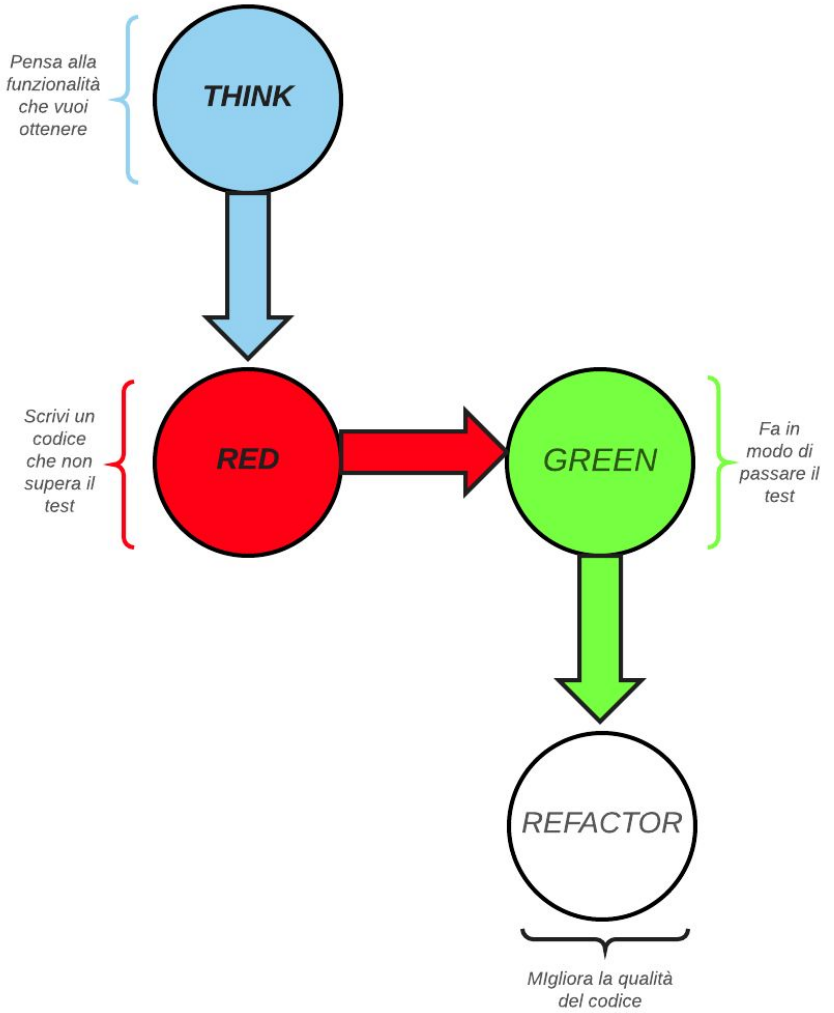
```
@Test
        public void testToString() { //Cambiare con il nome
della classe che avete creato
        String tester = new StringCalculator().toString();

        // assert statements
        assertEquals(true, tester.toString().equals(""), "Deve
restituire la stringa vuota");
    }
```

Si può scrivere anche così:

```
assertTrue(tester.toString().equals(""));
```

Esistono molte varianti di assert: assertTrue, assertFalse. assertEquals è il più flessibile, perchè overloadato su vari tipi primitivi, String e Object

Ciclo RGR esteso (reminder)



Primo test (RED)

- Scrivere prima il metodo che avete intenzione di sviluppare
- Scrivere il test in modo che richiami il metodo in modo sintatticamente corretto corretto (es: int calcola(String s))
- Deve essere un test semplice ed elementare (Happy Path). Esempio: fornendo una stringa vuota, bisogna ottenere 0
- Scrivetelo e mostratemi il codice e che genera un Rosso

Primo test (GREEN)

Il test che produce il maggior valore con il minimo sforzo è il calcolo di un singolo valore; meglio, una **classe** di valori (evitare il microtest)

Sviluppate il codice

Deve essere verde

SCAMBIATE I RUOLI!

Secondo test (RED)

Il metodo deve essere modificato.

Scrivere un test che ESTENDA le funzionalità del metodo; le funzionalità precedenti devono essere mantenute.

Show me the (RED) code.

Secondo test (GREEN)

Presumibilmente, test di due o più valori, stringa separata da un carattere di qualche tipo. Oppure semplicissimo: fornendo “1” il metodo deve restituire 1.

Sviluppate il codice

Deve essere verde

Refactoring time

Migliorate il codice…

Generalizzate (n numeri?)

Togliere rigidità (hardcoded values?)

Commenti (Javadoc?)

I test DEVONO RIMANERE VERDI

SCAMBIATE I RUOLI!

Terzo test (RED)

Fornite robustezza al vostro codice! Gestite errori del programmatore e/o dell'utente

null

Empty string

…

Show me the code(s)

Riferimenti

- Martin Fowler et al. (1999), Refactoring: Improving the Design of Existing Code, Addison-Wesley
- Robert C. Martin (2008), Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall
- Kent Beck (2000), Extreme Programming Explained: Embrace Change, 2nd Edition, Addison-Wesley

Possibili soluzioni (non-static)

```
@Test
public void parseOneNumber() {
StringCalculator calculator = new StringCalculator();
int result = calculator.calcola("1");
assertEquals(result, 1);}
@Test
public void parseTwoNumbers() {
StringCalculator calculator = new StringCalculator();
int result = calculator.calcola("1,2");
assertEquals(result, 3);}
```

GRAZIE!

CREDITS

Ringraziamenti a :

- Presentation template by SlidesCarnival
- Photographs by Unsplash
- ***marco.fracassi@trovaprezzi.it* & *roberto.grandi@trovaprezzi.it***
-
- ***Questo documento è distribuito con licenza CreativeCommon BY-SA 3.0***