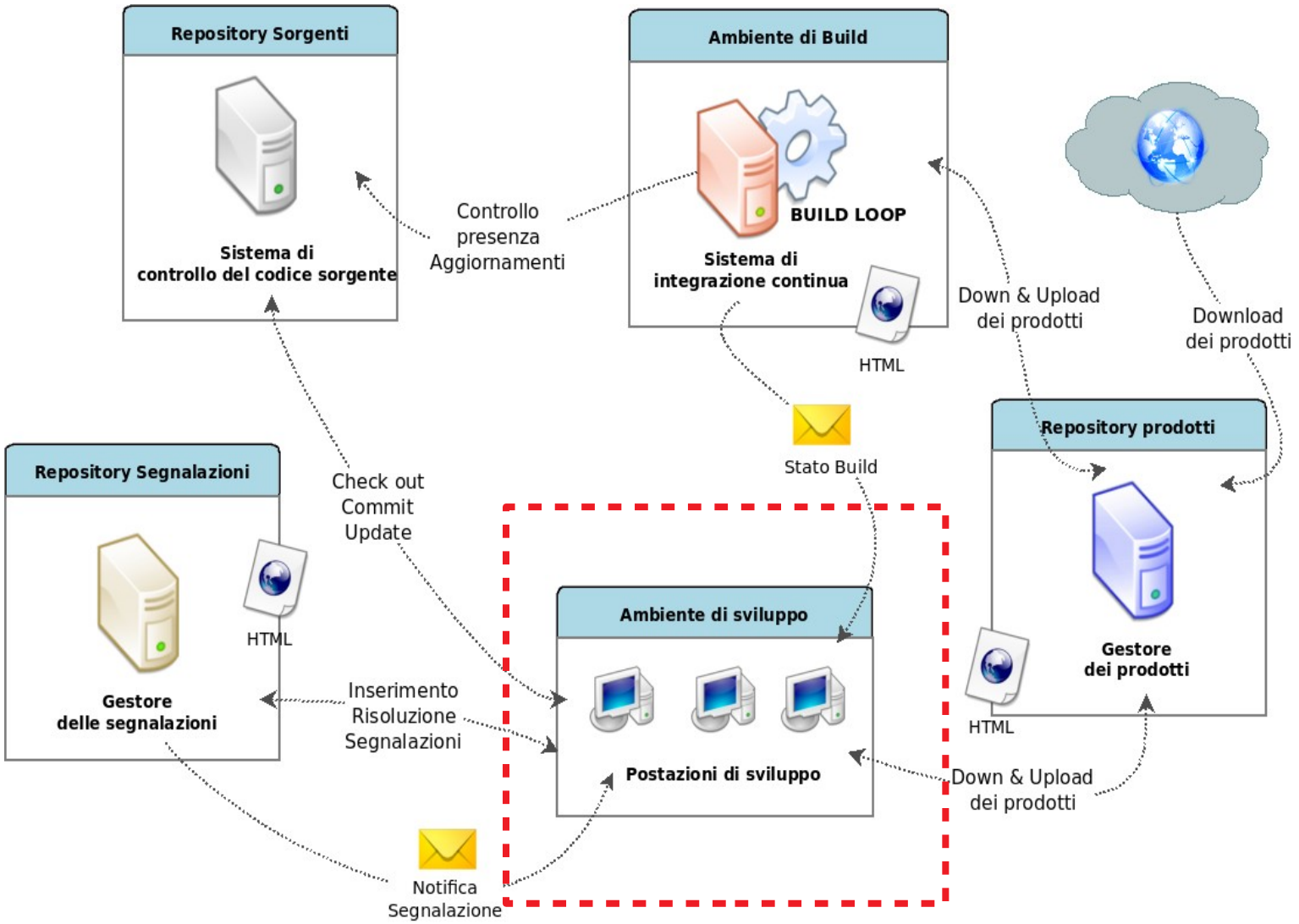
METODI E TECNOLOGIE PER LO SVILUPPO SOFTWARE

Nicola Bertazzo
nicola.bertazzo [at] unipd.it

Università degli Studi di Padova
Dipartimento di Matematica
Corso di Laurea in Informatica, A.A. 2021 – 2022

Visione Generale



Unit Testing

Definizione

In ingegneria del software, per unit testing (testing d'unità o testing unitario) si intende l'attività di testing (prova, collaudo) **di singole unità software**. Per unità si intende normalmente il **minimo componente** di un programma dotato di **funzionamento autonomo**; a seconda del paradigma di programmazione o linguaggio di programmazione, questo può corrispondere per esempio a una **singola funzione** nella programmazione procedurale, o una **singola classe** o un **singolo metodo** nella programmazione a oggetti.

...

Come le altre forme di testing, lo unit testing può variare da completamente "manuale" ad automatico. Specialmente nel caso dello unit testing automatico, lo sviluppo dei test case (cioè delle singole procedure di test) può essere considerato **parte integrante dell'attività di sviluppo** (per esempio, nel caso dello sviluppo guidato da test).

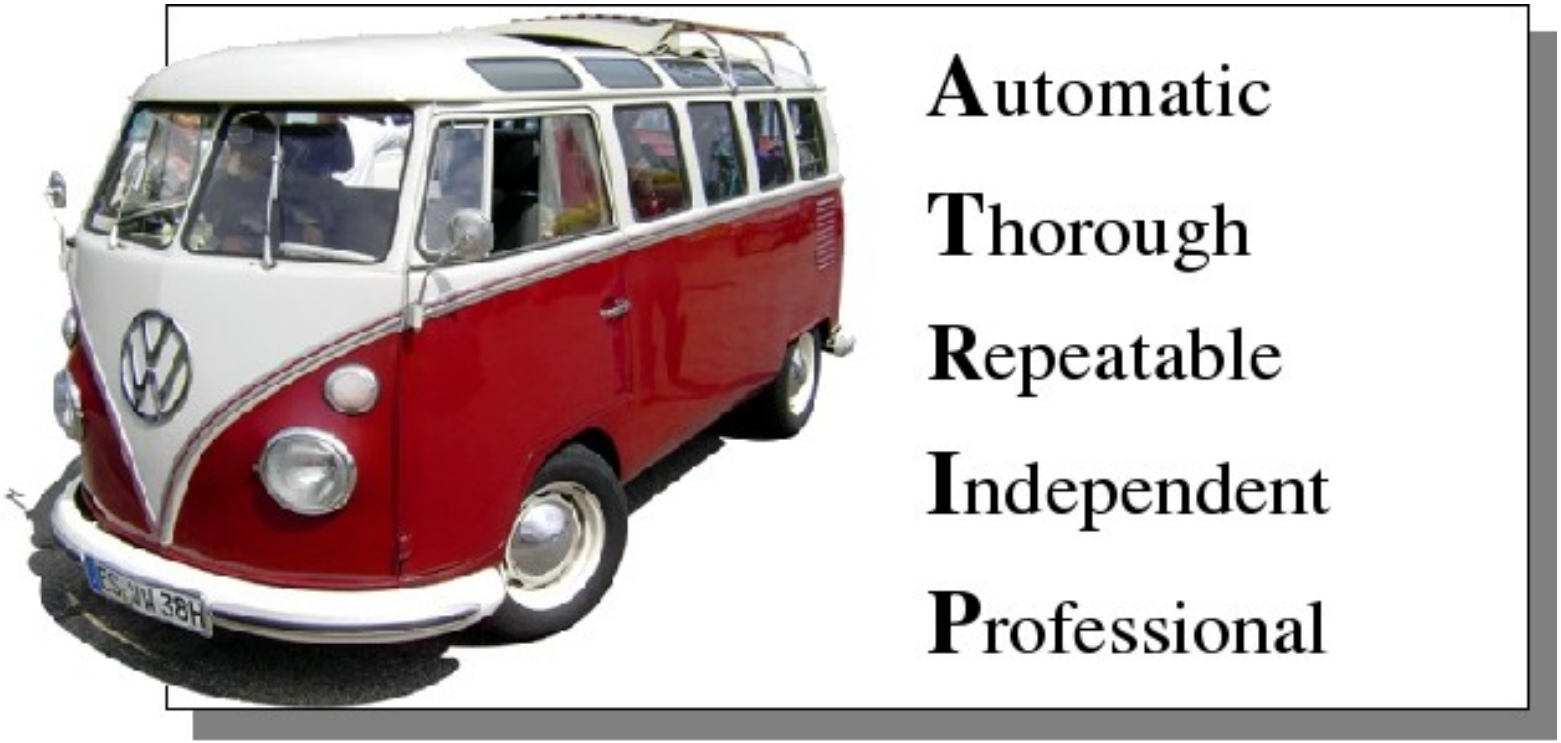
WIKIPEDIA
The Free Encyclopedia

Definizione

I **test di unità** sono del codice, **prodotto dallo sviluppatore**, che esercitano un'unità del programma.
Per unità si intende una **funzionalità atomica** che può essere verificata in modo isolato, in modo da assicurare che il risultato del test non sia influenzato da altre unità.
Nella programmazione ad oggetti un'unità può essere uno o più metodi di una classe, o un'istanza di una classe. Nella programmazione procedurale un'unità corrisponde ad una funzione.

Vengono **sviluppati dal programmatore che sviluppa le unità**, per **verificare l'assenza di alcuni errori**, e **documentare il comportamento dell'unità prodotta**.

Proprietà desiderabili (A TRIP)



Proprietà desiderabili (**A** TRIP)

Automatic

I test di unità devono essere eseguiti automaticamente. In ogni progetto deve essere disponibile un “automazione a comando” che permetta a tutti di invocare e far eseguire tutti o una parte dei test di unità in modo semplice.

Durante la fase di sviluppo del progetto è importante che i test possano eseguire:

- **In modo rapido**: I test di unità devono essere semplici e la loro esecuzione non deve impiegare più di pochi secondi.
- **Senza richiedere l’interazione umana**: Se un test di unità richiede che alcuni parametri siano inseriti, ogni volta, manualmente da uno sviluppatore, questo non permetterebbe di eseguire tutti i test del progetto in modo automatico a determinate ore del giorno.
- **In modo autonomo**: L’automazione che effettua l’esecuzione dei test di unità deve essere in grado di capire quando e dove i test falliscono ed avvisare gli sviluppatori. In questo modo gli sviluppatori saranno interrotti, dall’attività lavorativa, solo quando uno o più test falliranno.

***Thorough* (esaustivi)**

Dei buoni test di unità devono essere **esaustivi e accurat**i, devono verificare il comportamento di qualsiasi parte del progetto che potrebbe creare degli errori.

Esistono degli strumenti che permettono di misurare se ogni parte del progetto è stata eseguita durante la fase di test, e possono calcolare:

- La **percentuale di righe di codice** che vengono esercitate attraverso i test di unità nel progetto
- **La percentuale di possibili diramazioni** che vengono eseguiti dai test di unità
- **Il numero di eccezioni** che vengono controllate attraverso i test
- Altri dati che permettono di capire dove il progetto è carente di test di unità

Come si può intuire, non è detto che se in un progetto viene eseguito il 100% del codice dai test di unità questo è privo di errori.

Proprietà desiderabili (A TRIP)

Thorough (esaustivi)

jacoco > com.example.jacoco > Rectangle.java

Rectangle.java

```
1.  package com.example.jacoco;
2.
3.  public class Rectangle {
4.      private int x;
5.      private int y;
6.      private int width;
7.      private int height;
8.
9.      public Rectangle(int x, int y, int width, int height) {
10.         if (width <= 0 || height <= 0)
11.             throw new IllegalArgumentException("Dimensions are not positive");
12.
13.         this.x = x;
14.         this.y = y;
15.         this.width = width;
16.         this.height = height;
17.     }
18.
19.     public boolean intersects(Rectangle other) {
20.         if (x + width <= other.x)
21.             return false;
22.         if (x >= other.x + other.width)
23.             return false;
24.         return (y + height > other.y && y < other.y + other.height);
25.     }
26. }
```

Repeatable

I test di unità devono produrre sempre lo stesso risultato.

Per essere ripetibili, i test di unità devono avere l seguenti caratteristiche:

- **Essere indipendenti dall'ordine di esecuzione**: L'ordine di esecuzione dei test di unità non deve influenzare il risultato. Per questo è necessario che i test siano indipendenti
- **Essere indipendenti dall'ambiente di esecuzione**: L'esecuzione dei test non deve dipendere da risorse esterne al progetto o da risorse non gestite nel VCS. Se alcune unità devono utilizzare risorse esterne (p.es. database) è consigliato utilizzare la tecnica *Mock Object* per simulare il comportamento di queste componenti.

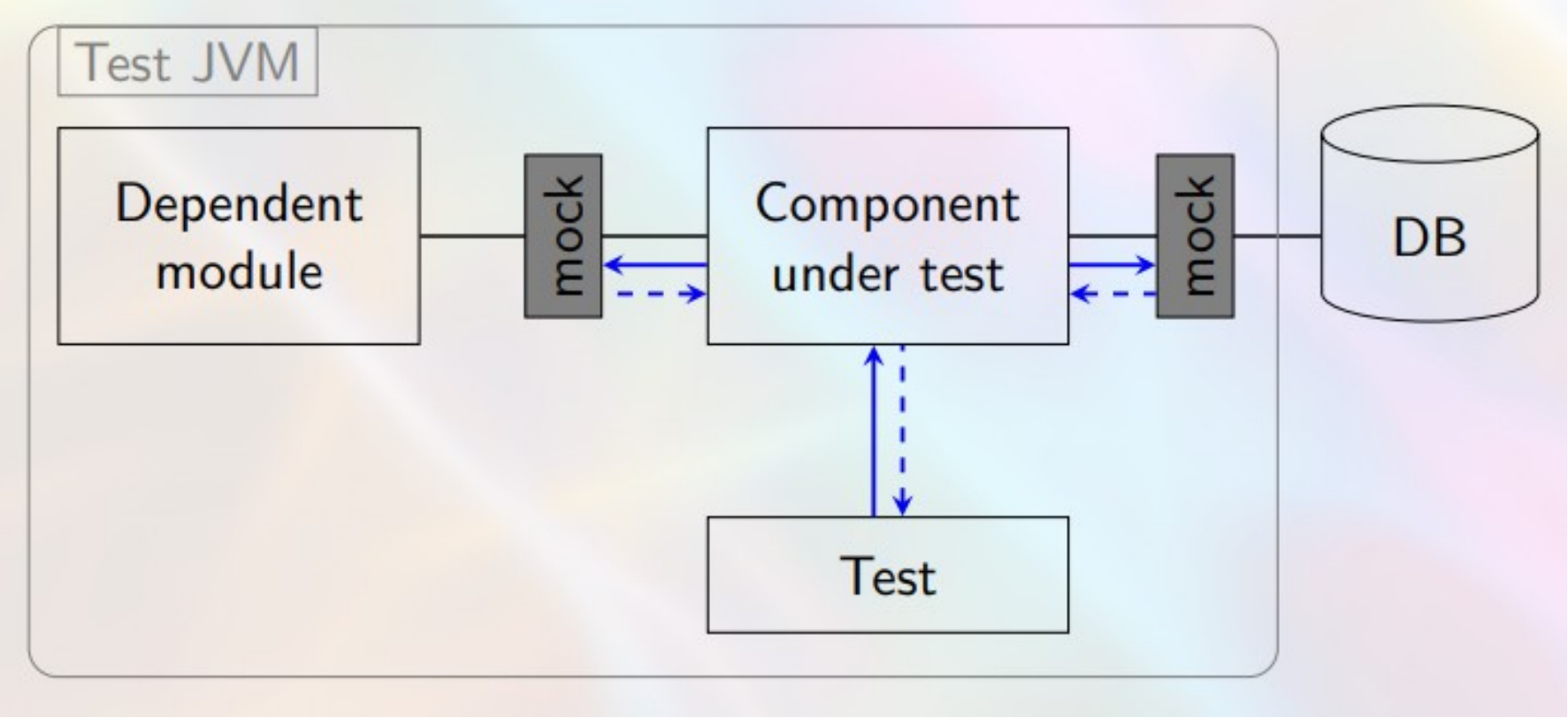
Independent

I test di unità devono essere il più possibile **indipendenti dall'ambiente di esecuzione, dagli elementi esterni al progetto e dall'ordine di esecuzione**. Quando si scrive un test è consigliato verificare il comportamento di un **singolo aspetto del progetto** (in questo modo si riesce ad identificare univocamente un'errore). Questo non significa che un test di un'unità deve avere solo una asserzione, ma deve controllare solo un metodo o più metodi che realizzano un aspetto di una funzionalità del progetto.

Se il test è indipendente il suo comportamento sarà ripetibile nel tempo, perché il suo comportamento non dipenderà dalle altre unità del progetto. La ripetibilità del test è un aspetto che permette di capire se il test è indipendente.

Proprietà desiderabili (A TRIP)

Independent



http://www.eclipsecon.org/europe2014/session/write-cool-scalable-enterprise-application-tests-xtend-embedded-dsls.html

Proprietà desiderabili (A TRI**P**)

Professional

Poiché i test di unità sono codice, devono essere **scritti e mantenuti con la stessa professionalità del codice di produzione** del progetto.

Visto che i buoni test di unità devono essere esaustivi, è ragionevole che il **numero di linee di codice** per realizzare i test sia **pari o a volte superiore** delle linee di codice in produzione.

Caratteristiche del framework

Per creare i test di unità sono necessari i seguenti componenti:

- Un modo per configurare l'ambiente di esecuzione del test
- Un modo per selezionare un test o un insieme di test da eseguire
- Un modo per analizzare i valori aspettati, prodotti dalle unità
- Un modo standard per eseguire ed esprimere se il test è stato superato, se è fallito o se sono stati prodotti degli errori

Caratteristiche del framework (JUnit 4)

Per creare i test di unità sono necessari i seguenti componenti:

- Un modo per configurare l'ambiente di esecuzione del test
  https://github.com/junit-team/junit4/wiki/Test-fixtures

- Un modo per selezionare un test o un insieme di test da eseguire
  https://github.com/junit-team/junit4/wiki/Aggregating-tests-in-suites
  https://github.com/junit-team/junit4/wiki/Categories

- Un modo per analizzare i valori aspettati, prodotti dalle unità
  https://github.com/junit-team/junit4/wiki/Assertions

- Un modo standard per eseguire ed esprimere se il test è stato superato, se è fallito o se sono stati prodotti degli errori
  https://github.com/junit-team/junit4/wiki/Getting-started

Cosa verificare?

- Are the Results **Right**?
- **B**oundary Conditions
- Check **I**nverse Relationships
- **C**ross-check Using Other Means
- Force **E**rror Conditions
- **P**erformance Characteristics

Are the Result Right?

Verificare se i risultati che essa produce sono corretti.

Per corretto si intende che il risultato atteso sia uguale al risultato prodotto dall'unità.

Capita che i requisiti non sono chiari, o possono cambiare nel tempo. In questi casi **i test di unità sono un buon punto di partenza per documentare nel codice**, come uno sviluppatore ha interpretato i requisiti e descrivere **il comportamento delle unità realizzate.**

Are the Result Right?

```
public class Calculator {
  public int evaluate(String expression) {
    int sum = 0;
    for (String summand: expression.split("\\+"))
      sum += Integer.valueOf(summand);
    return sum;
  }
}
```

Cosa verificare?

Are the Result Right?

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {
  @Test
  public void evaluatesExpression() {
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertEquals(6, sum);
  }
}
```

Cosa verificare?

Boundary Conditions (CORRECT)

Solitamente gli errori accadono in condizioni limite. Identificare le condizioni limite è una delle parti più importanti per creare delle buone unità

- **C**onformance - I valori sono conformi al formato atteso? [A+1] [1-2]
- **O**rdering - I valori seguono o non seguono un ordine? [++ 1 2 3]
- **R**ange - I valori sono all'interno di un valore di minimo e massimo appropriato? [2147483647 + 2]
- **R**eference - I valori possono provenire da codice che si riferisce a dati esterni che non sono sotto il controllo del codice?
- **E**xistence - I valori esistono (non sono nulli, non sono zero, sono presenti in un determinato insieme)? [null]
- **C**ardinality - I valori sono nella quantità desiderata? [1 + 2 + 3 +]
- **T**ime - I valori rispettano un ordine temporale?

Con il termine valore si fa riferimento sia ai parametri di input dei metodi di un'unità, che ai dati interni all'unità e ai risultati che questa produce.

Check Inverse Relationship

Alcune unità possono o devono essere verificati tramite l'applicazione della loro funzionalità inversa.

Esempi:

Calcolare la radice quadrata di un numero. Per testare se la radice quadrata è corretta è possibile elevare al quadrato il risultato ritornato dall'unità e confrontarlo con il dato di partenza.

Inserimento di un un elemento in una pila. Il modo più semplice per verificare se l'inserimento è andato a buon fine è quello di effettuare un prelevamento dalla pila e controllare che l'elemento ritornato sia l'elemento di partenza.

Cross-check Using Other Means

Utilizzare uno strumento esistente (oracolo) per verificare se la nuova unità ha lo stesso comportamento.

P.es. Migrazione da un vecchio sistema ad uno nuovo appena realizzato:

Si utilizza il vecchio sistema per verificare che quello nuovo abbia lo stesso comportamento.

Force error contitions

Nel mondo reale gli errori accadono. Una buona norma, per creare un buon progetto, è quello di ricreare le condizioni di errore e verificare che il progetto funzioni come ci si aspetta in queste condizioni.

Performance Caratteristics

I test di unità devono essere veloci perché devono poter essere eseguiti molto spesso.

Test Driven Development

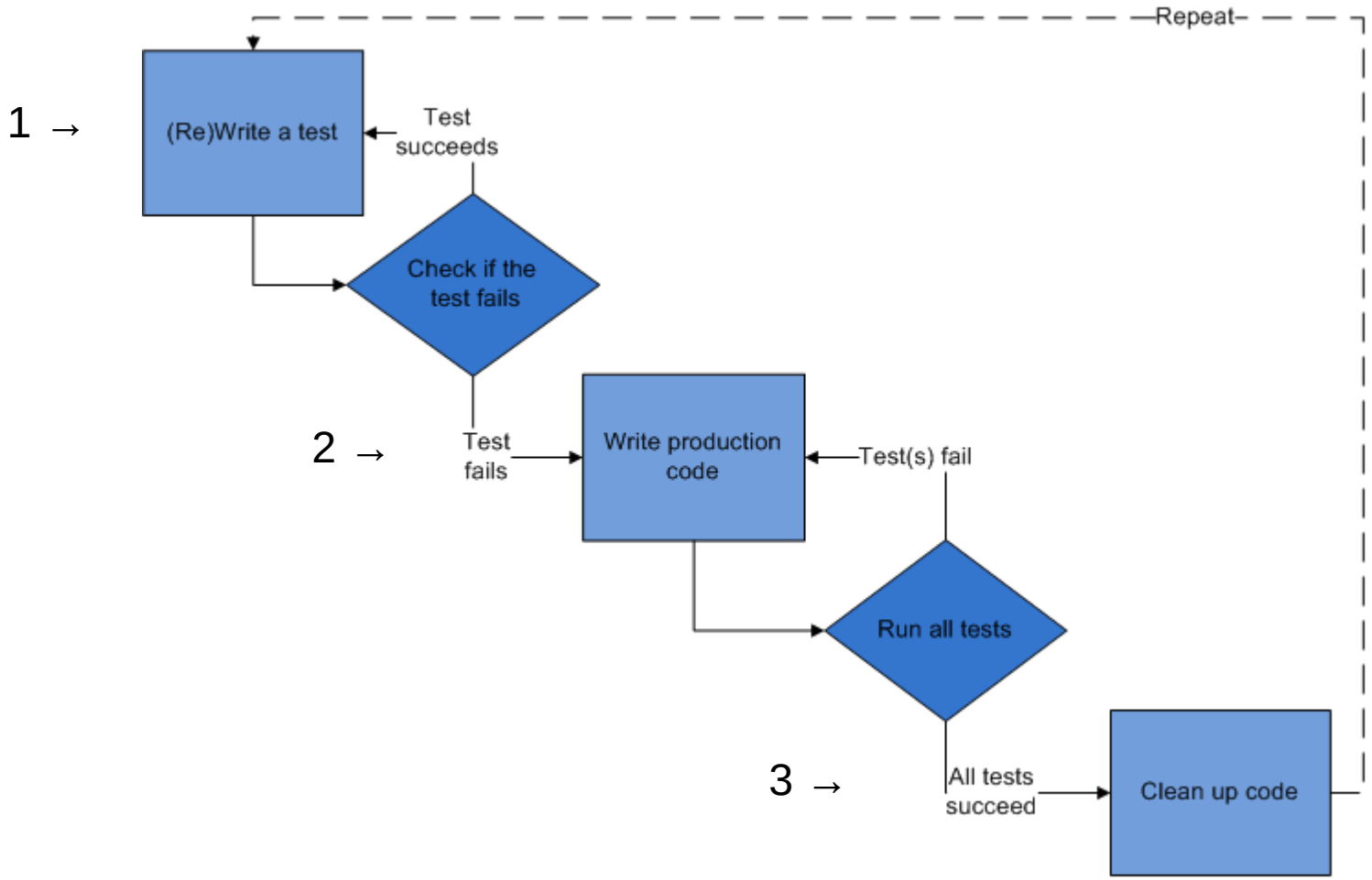
Il test-driven development (abbreviato in TDD), è un modello di sviluppo del software che prevede che la stesura dei test automatici avvenga prima di quella del software che deve essere sottoposto a test, e che lo sviluppo del software applicativo sia orientato esclusivamente all'obiettivo di passare i test automatici precedentemente predisposti.

Più in dettaglio, il TDD prevede la ripetizione di un breve ciclo di sviluppo in tre fasi, detto "ciclo TDD":

1) Nella prima fase (detta "fase rossa"), il programmatore scrive un test automatico per la nuova funzione da sviluppare, che deve fallire in quanto la funzione non è stata ancora realizzata.
2) Nella seconda fase (detta "fase verde"), il programmatore sviluppa la quantità minima di codice necessaria per passare il test.
3) Nella terza fase (detta "fase grigia" o di refactoring), il programmatore esegue il refactoring del codice per adeguarlo a determinati standard di qualità.

L'invenzione del metodo si deve a Kent Beck, uno dei padri dell'extreme programming (XP) e delle metodologie agili.

Test Driven Development



Test Driven Development

«Write new code only if an automated test has failed»
(Kent Beck, Test-Driven Development by Example)

«Only ever write code to fix a failing test»
(Lasse Koskela, Test Driven)

«We produce well-designed, well-tested, and well-factored code in small, verifiable steps»
(James Shore, Agile Development)

Test Driven Development

ESEMPIO

http://butunclebob.com/files/downloads/Bowling%20Game%20Kata.ppt

Riferimenti

https://it.wikipedia.org/wiki/Unit_testing
https://junit.org/junit4/
http://butunclebob.com/files/downloads/Bowling%20Game%20Kata.ppt
https://www.amazon.com/Pragmatic-Unit-Testing-Java-JUnit/dp/0974514012
https://www.amazon.it/Test-Driven-Development-Example-Kent-Beck/dp/0321146530