UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Ingegneria dell'Informazione ed Elettrica e Matematica applicata

Corso di Laurea in Ingegneria Informatica

APPUNTI DI INGEGNERIA DEL SOFTWARE

DI FRANCESCO PIO CIRILLO

https://github.com/francescopiocirillo

“Non ti abbattere mai”

DISCLAIMER

Questi appunti sono stati realizzati a scopo puramente educativo e di condivisione della conoscenza. Non hanno alcun fine commerciale e non intendono violare alcun diritto d'autore o di proprietà intellettuale.

I contenuti di questo documento sono una rielaborazione personale di lezioni universitarie, materiali di studio e concetti appresi, espressi in modo originale ove possibile. Tuttavia, potrebbero includere riferimenti a fonti esterne, concetti accademici o traduzioni di materiale didattico fornito dai docenti o presente in libri di testo.

Se ritieni che questo documento contenga materiale di tua proprietà intellettuale e desideri richiederne la modifica o la rimozione, ti invito a contattarmi. Sarò disponibile a risolvere la questione nel minor tempo possibile.

In quanto autore di questi appunti non posso garantire l'accuratezza, la completezza o l'aggiornamento dei contenuti p e non mi assumo alcuna responsabilità per eventuali errori, omissioni o danni derivanti dall’uso di queste informazioni. L’uso di questo materiale è a totale discrezione e responsabilità dell’utente.

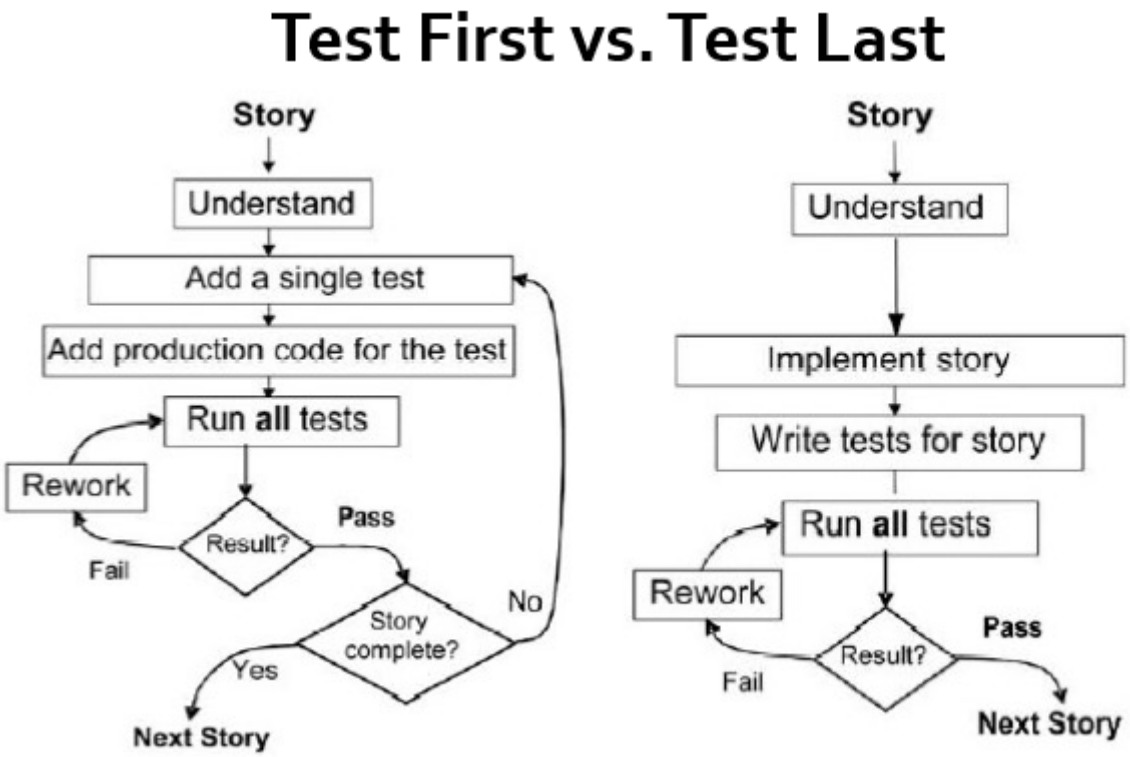
Testing

Sviluppo guidato dai Test (Test Driven Development)

Si tratta di una metodo di sviluppo software nel quale gli unit tests sono sviluppati prima del codice.

Come funziona:

1. Si **aggiunge un Test**: le user stories sono usate per comprendere chiaramente i requisiti
2. Si eseguono tutti i test e si vede quello nuovo (appena aggiunto) **fallire**: permette di assicurarsi che i test automatici funzionino correttamente e che il test aggiunto non passi per errore
3. Si scrive del codice: all'inizio il codice **ha il solo scopo di passare i test** e non devono essere aggiunte nuove funzionalità perché sarebbero non testate
4. **Si lanciano i test automatici**: se i test passano i programmatori possono essere sicuri che il loro codice rispetti tutti i requisiti testati
5. Refactoring: **si pulisce il codice** e si rieseguono i test per assicurarsi che il cleanup non abbia rotto nulla
6. Si ripete tutto

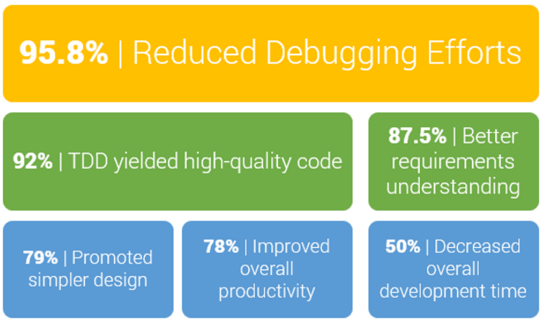


Benefici del Test Driven Development (TDD)

Il TDD favorisce una comprensione migliore dei tasks poiché scrivere i test prima di implementare una funzionalità richiede che il programmatore esprima la funzionalità in modo non ambiguo.

Il TDD favorisce un maggior focus su un singolo task visto che si avanza un Test Case alla volta e un solo test case ha scope limitato, questo permette al programmatore di scomporre features grandi in pezzi più piccoli e gestibili, riducendo il suo carico cognitivo.

Il TDD riduce lo sforzo per il rework, infatti visto che lo scope di un singolo test è limitato quando il test fallisce il **rework è più facile** e la causa è più facile da individuare. Inoltre, visto che il rework viene fatto a seguito di brevi raffiche di testing e implementazione il problema è ancora "fresco" nella mente del programmatore.



Automazione del Testing

Testing Manuale

Il testing manuale coinvolge un tester umano che fornisce input predefiniti al sistema usando l'interfaccia utente o un debugger.

Il Tester umano procede poi a comparare gli output generati dal sistema con gli oracoli attesi.

Quando il sistema si evolve il testing è da ripetere.

Il testing manuale può essere costoso e prono ad errori ma è comunque **fattibile per piccoli sistemi.**

Testing Automatico

Il testing automatico prevede la specificazione dei test cases in termini di:

- sequenza e timing degli input
- la traccia degli output attesi

L'infrastruttura di test esegue automaticamente i test cases e compara l'output del sistema e la traccia degli output attesi (expected output trace).

I benefici del testing automatico sono:

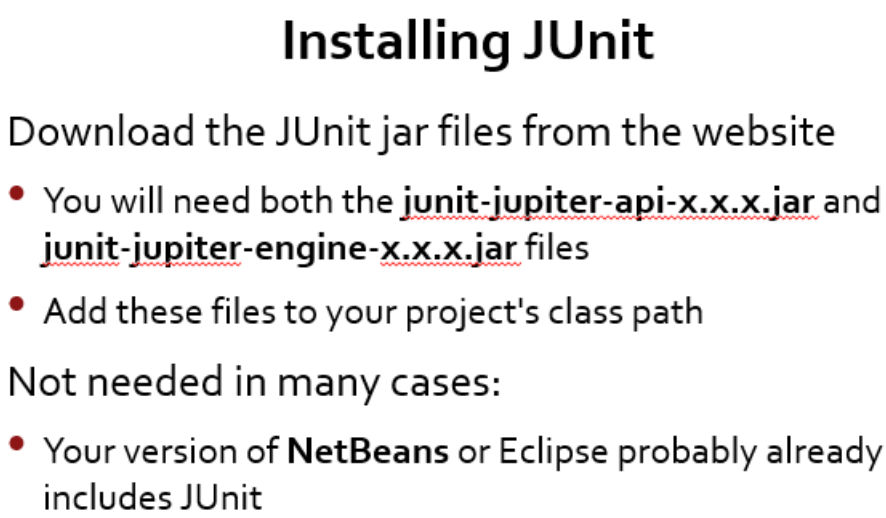
- i test sono ripetibili e i costi scendono sostanzialmente
- una volta che un fault (difetto) è corretto i test possono essere rapidamente ripetuti

- **incoraggia il Refactoring** perché gli sviluppatori sono più sicuri di poterlo fare senza introdurre nuovi bug nel mentre

JUnit

> 💡 **JUnit è lo standard de facto** per l'automazione del testing in Java, è portabile su ogni piattaforma e usabile da linea di comando ma anche con supporto diretto da IDE come Netbeans o Eclipse.

Sito: https://junit.org/junit5/



Usare JUnit

Si definisce una **classe di test per ogni classe** che si vuole testare, il nome della test class deve essere per **convenzione** il nome della classe da testare con Test aggiunto alla fine. (Adder → AdderTest).

La tested class deve essere pubblica, se la classe testata ha metodi protected o package la test class deve essere nello stesso package.

Nella test class devono esserci i seguenti import:

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
```

Supponiamo di voler creare una classe di test per la classe Adder

```
public class AdderTest {
  // put here the test methods
}
```

Nella classe di test si devono inserire metodi di test per effettuare i veri e propri test, la convenzione è che per testare methodName() si definisce testMethodName(). I metodi di test sono public void e non hanno parametri.

```
@Test
public void testAdd() {
   // implementation of a test for method ‘add’
}
```

L’annotazione **@Test** indica a JUnit che testAdd è un test method, infatti la classe di test potrebbe anche contenere metodi non di test.

I metodi di test chiamano uno o più metodi statici della **classe Assert** per specificare le condizioni che devono essere verificate:

```
assertEquals(expected, actual) //or
assertArrayEquals(expected, actual) //requires that ‘expected’ is equal to ‘actu
assertTrue(cond) //requires that `cond’ is true
assertFalse(cond) //requires that ‘cond’ is false
assertNull(obj) //requires that ‘obj’ is null
assertNotNull(obj) //requires that ‘obj’ is NOT null
```

Un test si ritiene passato se tutte le condizioni al suo interno sono verificate.

Esempio di Test Driven Development

Supponiamo di voler costruire una classe che somma interi, seguendo i principi del TDD come prima cosa definiamo l’interfaccia della classe:

```
public class Adder {
    public int add(int a, int b) {
        // empty implementation, to
        // make the class compilable!
        return 0;
    }
}
```

Ora definiamo una classe di test per la classe Adder e un **test method** per il metodo add:

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class AdderTest {

    @Test
    public void testAdd() {
        // test code goes here
```

```
    }
}
```

A questo punto possiamo scrivere codice di test usando le asserzioni:

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class AdderTest {

    @Test
    public void testAdd() {
        Adder a = new Adder();
        assertEquals(7, a.add(4, 3));
        assertEquals(5, a.add(10, -5));
    }
}
```

La classe di test a questo punto è completa e compilabile.

Risultati

Nei test reports JUnit usa due diversi termini per quando le cose vanno male:

- **failure**: il test è stato eseguito ed almeno una assertion si è rivelata false
- **error**: un errore (come ad esempio una exception) ha impedito l'esecuzione del test

Timeout

Si può specificare che un metodo di test deve considerarsi fallito se non completato entro un tempo limite usando come parametro dell'annotazione **@Test time-out=tempo** in millisecondi

```
@Test(timeout = 3000) // 3 seconds
public void testAdd() {
    // implementation of the test
}
```

Exceptions

Si può specificare che un metodo di test debba per forza lanciare una eccezione (se l'eccezione NON viene lanciata il test fallisce) sempre usando un parametro dell'annotazione @Test

```
@Test(expected = NullPointerException.class)
public void whenExceptionThrown_thenExpectationSatisfied() {
    String test = null;
    test.lenght();
}
```

Fixtures (apparecchi)

Spesso molti test hanno bisogno di creare gli stessi oggetti, se questi vengono creati in ogni metodo di test avremo del codice duplicato.

> 💡 Per evitarlo in JUnit si può creare **una Fixture** cioè un insieme di oggetti (conservati in variabili di istanza della classe di test) reinizializzati sempre nello stesso modo prima di lanciare ogni metodo di test.

L'annotazione **@BeforeEach** permette di specificare che un metodo deve essere chiamato prima di ogni test, in questo modo possiamo inizializzare una Fixture.

L'annotazione **@AfterEach** permette di specificare che un metodo deve essere chiamato dopo ogni test, in questo modo si possono ad esempio rilasciare delle risorse o chiudere file o altro.

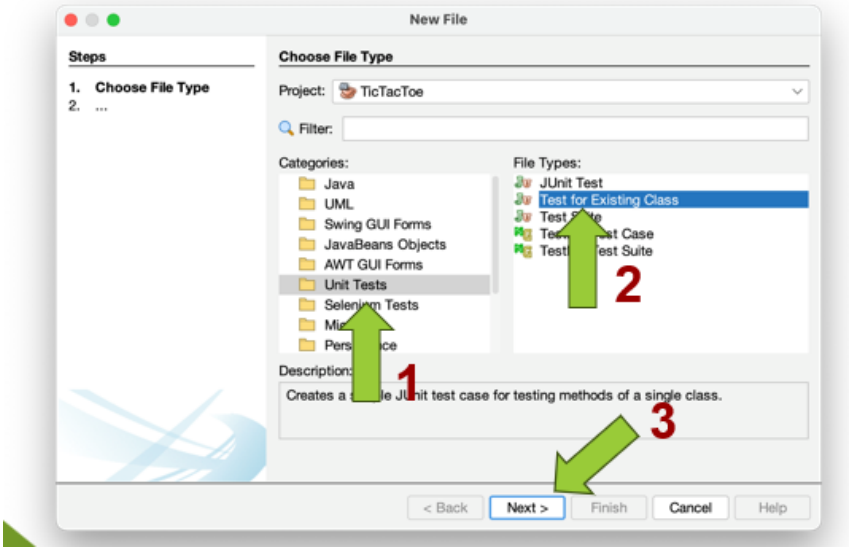
```
public class AdderTest {

    private Adder a;

    @BeforeEach
    public void setUp() {
        a = new Adder();
    }

    @Test
    public void testAddPositive() {
        assertEquals(7, a.add(4, 3));
    }

    @Test
    public void testAddNegative() {
        assertEquals(5, a.add(10, -5));
    }
}
```

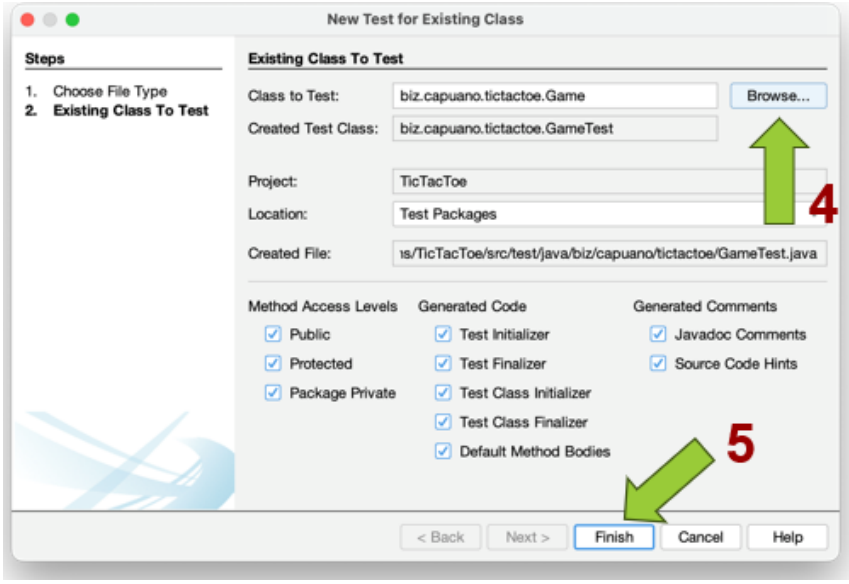
Tips

- **Testa una cosa alla volta** per metodo di test
- I test dovrebbero **evitare l'utilizzo di logica** (if, loop ecc) se possibile
- I test dovrebbero **evitare l'utilizzo di try/catch**: se ci si aspetta un'eccezione questa dovrebbe andare nel parametro di @Test, altrimenti dovrebbe essere gestita da JUnit

Integrazione con Netbeans

On your project Click on "**New File**" and then...





Select the class to test

Una nuova classe di test è aggiunta al package di test

Bisogna personalizzare le classi di test in modo da poter davvero testare le proprie classi

Per eseguire i test si clicca Run e poi Test Project