8. Introduzione al Testing del Software e Unit Testing in Java con JUnit 5

Questo capitolo introduce i concetti fondamentali del testing del software, con un focus particolare sullo unit testing in Java, utilizzando il framework JUnit 5 e la metodologia Test-Driven Development (TDD).

1.1. Introduzione al Testing del Software

Il testing del software è un'attività cruciale nel ciclo di vita dello sviluppo, volta a garantire la **qualità** e la **correttezza** di un programma. Non si tratta solo di trovare errori, ma di fornire informazioni agli stakeholder sulla qualità del prodotto e di ridurre i rischi futuri.

1.1.1. Concetti Preliminari: Errore, Difetto (Bug) e Fallimento (Anomalia)

Nel contesto del testing, è fondamentale distinguere tra questi tre termini correlati ma distinti:

- **Fallimento (Problema, Anomalia)**: Si riferisce a una **differenza** osservata tra un **risultato attuale** del software e il risultato **atteso**. È la manifestazione esterna di un problema.
- **Falla (Difetto, Bug)**: È la **causa** interna di un'anomalia. Un singolo bug può causare diverse anomalie, e un fallimento può essere il risultato di più bug. Un bug può rimanere "latente" se non si manifesta in un fallimento.
- **Errore**: L'**azione umana** che ha portato all'introduzione di una falla (bug) nel software, ad esempio una distrazione del programmatore.

La progressione logica è quindi: **Errore → Difetto/Bug → Fallimento**.

1.1.2. Verifica vs. Validazione

Questi due concetti definiscono gli **obiettivi** principali del testing:

- **Verifica**: Si concentra sul "costruire il software nel modo giusto" ("Have we built the thing right?"). Consiste nel **controllare la correttezza** del software rispetto alle sue **specifiche**. Se un progetto non ha specifiche chiare, non può essere "giusto" o "sbagliato", ma solo "sorprendente". Tuttavia, le specifiche stesse potrebbero essere incomplete o errate.
  - Abbiamo scritto la cosa nel **modo giusto?**
- **Validazione**: Si concentra sul "costruire il software giusto" ("Have we built the right thing?"). Riguarda il controllo dell'**adeguatezza del software** rispetto alle **aspettative** e alle esigenze degli stakeholder (clienti, utenti finali, ecc.).
  - Abbiamo scritto la **cosa giusta?**

1.1.3. Definizioni di "Software Testing"

Diverse definizioni evidenziano la complessità e l'importanza del software testing:

- **Cem Kaner/Wikipedia**: "Un'indagine condotta per fornire agli stakeholder informazioni sulla qualità del prodotto o servizio software sotto test."
- **ISO/IEC/IEEE 24765:2010**: "Attività in cui un sistema o componente viene eseguito in condizioni specificate, i risultati vengono osservati o registrati, e viene fatta una valutazione di qualche aspetto del sistema o componente."
- **ISTQB (International Software Testing Qualifications Board)**: "Il processo che consiste in tutte le attività del ciclo di vita, sia statiche che dinamiche, relative alla pianificazione, preparazione e valutazione dei prodotti software e dei prodotti di lavoro correlati per determinare che essi soddisfino i requisiti specificati, per dimostrare che sono idonei allo scopo e per rilevare difetti."
- **Definizione Generale**: "Il processo complessivo di pianificazione, preparazione ed esecuzione di una suite di diversi tipi di test progettati per validare un sistema in sviluppo, al fine di raggiungere un livello accettabile di qualità e di evitare rischi inaccettabili."

1.1.4. Elementi Chiave del Testing

Il testing è un'attività complessa che coinvolge diversi aspetti:

- **Attività/Processo**: Include la raccolta dei requisiti, la preparazione dei test, l'esecuzione dei test e la valutazione dei risultati.
- **Varietà di Tipologie**: Esistono diverse forme di testing, come il testing manuale, il testing automatico, il testing di unità e il testing di integrazione.
- **Verifica e Validazione**: Si possono formalizzare dei "criteri di accettazione" per valutare il software.
- **Riduzione dei Rischi**: Il testing aiuta a prevenire problemi futuri nel software.
- **Vari Stakeholder**: Diversi ruoli sono coinvolti, tra cui programmatori, tester, clienti e integratori.

1.2. Unit Testing in Pratica

Lo unit testing è una pratica di testing che si concentra sulla **verifica di singole unità funzionali** del software in isolamento. In Java, una **"unità"** è tipicamente una **singola classe**.

1.2.1. Concetto di Unit Testing

Quando si esegue lo unit testing, si testa una singola classe (l'unità sotto test, o UUT - Unit Under Test). Per una classe `SomeClass`, si scriverà una classe `SomeClassTest` che conterrà metodi di test specifici per esercitare la funzionalità offerta da `SomeClass`. Lo unit testing si distingue dall'integration testing, che invece verifica la corretta interazione tra diverse unità.

1.2.2. Automatizzazione dei Test

Inizialmente, i test possono essere eseguiti manualmente, ma questo processo è inefficiente e soggetto a errori. L'automatizzazione dei test permette di eseguire un gran numero di test case in modo rapido e ripetibile.

Un esempio di automatizzazione rudimentale in Java potrebbe prevedere l'uso di `if` per confrontare i risultati attesi con quelli effettivi, ma questo approccio soffre di problemi di qualità del codice (mancanza di isolamento, ripetizioni).

1.2.3. JUnit 5: Un Framework per lo Unit Testing in Java

JUnit 5 è un framework ampiamente utilizzato per scrivere e eseguire test unitari in Java. Offre un'API ricca e una struttura ben definita per organizzare i test.

1.2.3.1. Struttura di un Test con JUnit 5

Una classe di test JUnit è una "test suite" che raccoglie i "test case" associati a una specifica UUT.

- **Annotazioni**: Diciture come `@Test` e `@BeforeEach` sono annotazioni. Le annotazioni in Java forniscono informazioni aggiuntive ai costrutti del programma e possono essere analizzate a tempo di compilazione o di esecuzione da altri componenti software. Possono avere parametri e possono essere applicate a dichiarazioni di classe, campi, metodi, ecc.
  - `@Test`: Indica che un metodo è un test case.
  - `@BeforeEach`: Indica che un metodo deve essere eseguito prima di ogni test case per gestire il contesto (es. inizializzare la UUT).
  - `@BeforeAll`, `@AfterAll`: Applicate a metodi statici, vengono eseguiti una sola volta prima o dopo l'esecuzione di *tutti* i test della classe.
  - `@AfterEach`: Applicata a un metodo d'istanza, viene eseguita dopo ogni test case.
- **Asserzioni**: La classe `org.junit.jupiter.api.Assertions` fornisce vari metodi per esprimere asserzioni, **confrontando un risultato atteso con un risultato effettivo**. Esempi includono `assertEquals(expected, actual)`, `assertTrue(condition)`, `assertFalse(condition)`, `assertNull(object)`, `assertNotNull(object)`, `assertArrayEquals(expectedArray, actualArray)`. È possibile importare i metodi statici di `Assertions` per un uso più conciso (`import static org.junit.jupiter.api.Assertions.*;`).
- **Ciclo di Vita dei Test**: Di default, JUnit crea una nuova istanza della classe di test prima di invocare ogni metodo di test. Le annotazioni `@BeforeAll`, `@AfterAll`, `@BeforeEach`, `@AfterEach` permettono di agganciarsi alle varie fasi del ciclo di vita dei test.

1.2.3.2. Esempio di Classe di Test (BuggyNumFinderJUnitTest)

Consideriamo un esempio di test per una classe `BuggyNumFinder` che trova il più piccolo e il più grande elemento in un array:

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
```

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

public class BuggyNumFinderJUnitTest {

    BuggyNumFinder numFinder; // La Unit Under Test (UUT)

    @BeforeEach
    void setup() {
        System.out.println("Setup");
        numFinder = new BuggyNumFinder(); // Inizializza la UUT prima di ogni test
    }

    @Test
    public void someNumbers() {
        int[] input1 = new int[]{4, 25, 7, 9};
        numFinder.find(input1);
        Assertions.assertEquals(4, numFinder.smallest); // Asserisce che il più piccolo sia 4
        Assertions.assertEquals(25, numFinder.largest); // Asserisce che il più grande sia 25
    }
}
```

1.2.3.3. Separazione dei Sorgenti di Test

È una buona pratica tenere separati i sorgenti di test da quelli dell'applicazione principale. La convenzione consolidata è:

- **Sorgenti dell'applicazione principale**: sotto `src/main/java/`
- **Sorgenti di test**: sotto `src/test/java/`

Strumenti di build come Maven e Gradle utilizzano questa convenzione. Inoltre, le classi di test sono spesso dichiarate nello stesso package delle classi di produzione per accedere ai membri `package-private`.

1.2.3.4. Architettura di JUnit 5

JUnit 5 è modulare e si compone di tre parti principali:

- **JUnit Platform**: La piattaforma comune per l'esecuzione dei test attraverso l'astrazione di un "engine". Include l'API usata dai tool di build e dagli IDE (es. Gradle interagisce con questo componente).
- **JUnit Jupiter**: L'API per scrivere test in JUnit 5 (`junit-jupiter-api`) e il relativo engine (`junit-jupiter-engine`).
- **JUnit Vintage**: Un engine per l'esecuzione di test scritti con JUnit 4 (`junit-vintage-engine`), garantendo la retrocompatibilità.

1.2.3.5. Pattern AAA (Arrange - Act - Assert)

Questo è un pattern tipico per organizzare i metodi di test, rendendoli chiari e leggibili:

1. **Arrange (Preparazione)**: Imposta la Unit Under Test (UUT) e il contesto del test (es. inizializzazione degli oggetti, configurazione dei dati).
2. **Act (Azione)**: Esegue la funzionalità che si vuole testare (es. chiama un metodo sulla UUT).
3. **Assert (Verifica)**: Asserisce l'aspettativa rispetto al risultato effettivo, utilizzando i metodi di asserzione di JUnit.

Esempio:

```
// Arrange: imposta la UUT e il contesto del test
String s = "hello world";
String substring = "world";

// Act: esercita la funzionalità
boolean contained = s.contains(substring);

// Assert: asserisce l'aspettativa rispetto al risultato effettivo
Assertions.assertTrue(contained);
```

1.3. Test-Driven Development (TDD)

Il Test-Driven Development (TDD) è una metodologia di sviluppo software in cui i **test** non sono solo un'attività post-implementazione, ma **guidano l'intero processo** di progettazione e implementazione. Si basa su un ciclo iterativo noto come **RED-GREEN-REFACTOR**.

1.3.1. Il Ciclo RED-GREEN-REFACTOR

1. **RED (Scrivi un test che fallisce)**: Si scrive un test per una funzionalità che si intende implementare. Poiché la funzionalità non esiste ancora o non è completa, questo test è destinato a fallire. Questo passaggio forza il programmatore a pensare ai requisiti e all'interfaccia della funzionalità prima di scrivere il codice di produzione.
2. **GREEN (Fai passare il test)**: Si scrive il codice di produzione minimo indispensabile per far sì che il test precedentemente fallito ora passi. L'obiettivo qui non è la perfezione del codice, ma la funzionalità.
3. **REFACTOR (Migliora il codice)**: Una volta che il test passa, si può migliorare il codice (sia della UUT che del test stesso) senza alterarne la funzionalità. Durante questa fase, si eseguono nuovamente tutti i test per assicurarsi di non aver introdotto regressioni (difetti in componenti che prima funzionavano correttamente).

Questo ciclo viene ripetuto per ogni incremento di funzionalità.

1.3.2. Esempio di TDD: La Classe `Tv`

Vediamo un esempio pratico del ciclo TDD per una semplice classe `Tv` con funzionalità di accensione/spegnimento.

1.3.2.1. Step 1: RED (Scrivi il test che fallisce)

Si scrive un test per la funzionalità `turnOn()` della classe `Tv`, che non è ancora implementata correttamente.

```
// Classe Tv (implementazione iniziale minima per compilare)
public class Tv {
    public void turnOn() { }
    public boolean isOn() { return false; }
}

// Classe TvTest
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Assumptions; // Usato per saltare test se una precondizione non è soddisfatta

public class TvTest {
    @Test
    public void testTurnOnWhenOff() {
        Tv tv = new Tv();
        // Assumiamo che la TV sia spenta prima di accenderla
        Assumptions.assumeTrue(!tv.isOn());
        tv.turnOn();
        // Asseriamo che la TV sia accesa dopo averla accesa
        Assertions.assertTrue(tv.isOn());
    }
}
```

Eseguendo questo test, ci si aspetta che fallisca (RED), perché `tv.isOn()` restituisce sempre `false`.

1.3.2.2. Step 2: GREEN (Fai passare il test)

Si implementa la funzionalità `turnOn()` e `isOn()` nella classe `Tv` in modo che il test passi.

```
// Classe Tv (implementazione per far passare il test)
public class Tv {
    boolean on; // Stato della TV

    public void turnOn() {
```

```
        this.on = true; // Accende la TV
    }

    public boolean isOn() {
        return this.on; // Restituisce lo stato attuale
    }
}
```

Eseguendo i test ora, dovrebbero passare (GREEN).

1.3.2.3. Step 3: REFACTOR (Migliora il codice)

Si migliora il codice, ad esempio aggiungendo un costruttore che inizializzi lo stato `on` a `false`. Si rieseguono i test per assicurarsi che non siano state introdotte regressioni.

```
// Classe Tv (dopo il refactoring)
public class Tv {
    boolean on;

    public Tv() {
        this.on = false; // Inizializza la TV spenta
    }

    public void turnOn() {
        this.on = true;
    }

    public boolean isOn() {
        return this.on;
    }
}

// Classe TvTest (con setup migliorato)
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Assumptions;

public class TvTest {
    Tv tv; // Dichiarazione della UUT

    @BeforeEach
    public void setUp() {
        tv = new Tv(); // Inizializzazione della UUT prima di ogni test
    }

    @Test
    public void testTurnOnWhenOff() {
        Assumptions.assumeTrue(!tv.isOn());
        tv.turnOn();
        Assertions.assertTrue(tv.isOn());
    }
}
```

Questo conclude un'iterazione del ciclo TDD. Per aggiungere nuove funzionalità (es. `turnOff()`, gestione dei canali), si ripartirebbe dal passo RED.

1.4. Qualità Interna del Software e Refactoring

Il TDD, insieme ad altri principi e pratiche di progettazione, contribuisce a migliorare la **qualità interna** del software, che si riferisce a quanto il codice è ben costruito, leggibile, flessibile e riusabile.

1.4.1. Qualità Interna vs. Qualità Esterna

- **Qualità Esterna**: Si riferisce agli aspetti **funzionali** (il software fa ciò che deve) e **non funzionali** (performance, sicurezza, usabilità) percepiti dall'utente.
- **Qualità Interna**: Riguarda la **struttura** e la **leggibilità** del codice, la sua **manutenibilità**, **flessibilità** e **riusabilità**. Un software con buona qualità interna è meno "costoso" da mantenere nel breve, medio e lungo termine.

1.4.2. Refactoring

Il **refactoring** è un'operazione di modifica del codice che **non aggiunge nuove funzionalità**, ma ha lo scopo di migliorare la qualità interna del software (formattazione, espressività semantica, struttura). È fondamentale per preparare il codice a futuri cambiamenti (agilità).

- **Necessità del Refactoring**: Una buona progettazione non si ottiene al primo tentativo. Il refactoring è una pratica necessaria e supportata da test automatici (che verificano l'assenza di regressioni) e design pattern (che forniscono "ricette" per una buona costruzione o rifattorizzazione).

1.5. Strumenti per lo Sviluppo Java

Per lo sviluppo di progetti Java, vengono utilizzati diversi strumenti:

- **Java Development Kit (JDK)**: Include la Java Runtime Environment (JRE), che contiene la Java Virtual Machine (JVM) e le Java Class Libraries (JCL), oltre a strumenti di sviluppo come `javac` (compilatore) e `jshell`.
- **Terminali**: Per l'accesso al file system e ai programmi da linea di comando.
- **Gradle**: Un sistema di build che automatizza la compilazione, il testing e il packaging del software.
- **Git**: Un sistema di controllo versione per tracciare le modifiche al codice e collaborare.
- **JUnit**: Il framework per lo unit testing, come discusso in precedenza.
- **Visual Studio Code**: Un ambiente di sviluppo integrato (IDE) che facilita la scrittura, il debug e l'esecuzione del codice.