UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
FACOLTA' DI INGEGNERIA

CORSO DI LAUREA INGEGNERIA INFORMATICA
TESI DI LAUREA IN INGEGNERIA DEL SOFTWARE

STRUMENTI PER IL TESTING DELLE GUI

Relatore
Ch.mo. Prof. Porfirio Tramontana

Candidato
Fernando Di Costanzo

Matricola
N46002743

Anno Accademico 2017 - 2018

UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato finale in **Ingegneria del Software**

Strumenti per il testing delle GUI

Anno Accademico 2017/2018

Candidato:
Fernando Di Costanzo

Matricola:
N46002736

"Il test di un programma può essere usato per mostrare la presenza di bug, ma mai per mostrare la loro assenza."

-E.W. Dijkstra

Indice

Introduzione 5
Capitolo 1 Testing 7
1.1 Testing unitario 7
Capitolo 2 Applicazione 10
Capitolo 3 JUnit3 13
3.1 Funzionamento 14
3.2 Esempio Applicativo JUnit3 15
3.3 Limiti di JUnit3 19
Capitolo 4 JUnit4 20
4.1 JUnit4 vs JUnit3 20
4.2 Esempio Applicativo JUnit4 23
4.3 Limiti di JUnit4 26
Capitolo 5 Maveryx 27
5.1 Eliminazione dei GUI Maps 27
5.2 Esempio Applicativo 28
Capitolo 6 JUnit5 29
6.1 JUnit5 VS JUnit4 29
6.2 Esempio Applicativo 31
Conclusioni 35
Bibliografia 36

Introduzione

GUI è l'acronimo di Graphical User Interface. Un utente può interagire con un computer (o qualsiasi software o sito web su un computer) attraverso due tipi di interfacce. Queste interfacce sono:

- **interfaccia a riga di comando**
- **interfaccia utente grafica.**

L'interfaccia della riga di comando è piuttosto difficile rispetto all'interfaccia grafica, poiché l'utente è tenuto a ricordare i comandi e i diversi tipi di sintassi. L'interfaccia utente grafica prevede l'interazione con elementi quali casella di testo, pulsanti di opzione, menu a discesa, caselle di controllo, ecc. Che sono visibili sullo schermo.

Durante l'interazione con la GUI, l'utente non è tenuto a ricordare i comandi, ma può semplicemente interagire tramite il puntatore del mouse e selezionare qualsiasi elemento sul sito Web o sull'applicazione Windows.

La GUI è spesso progettata per l'utente ingenuo che non ha la conoscenza dei comandi ma può interagire con il puntatore del mouse e interagire con gli elementi web.

Il test della GUI dovrebbe concentrarsi sul design dello schermo e sulla facilità dell'interazione con gli elementi grafici sullo schermo. Inoltre, il tester deve tenere a mente la reattività e l'aspetto dello schermo durante lo svolgimento del test della GUI. La conclusione è che il tester deve pensare come un utente perché l'utente finale non ha alcuna conoscenza dell'applicazione o del software, ma è l'interfaccia utente del software che fa credere all'utente l'utilità di tale applicazione.

Il test della GUI può essere eseguito nei seguenti tre modi:

- **Test GUI manuale:** come qualsiasi approccio tradizionale di test manuale, questo approccio è molto semplice in cui le schermate grafiche vengono controllate manualmente dal tester e confrontate con le schermate del prototipo oi casi di test come preparati rispetto ai documenti dei requisiti aziendali.

- **Record and Replay (Test Automation):** possiamo automatizzare la verifica della GUI con l'aiuto di strumenti come QTP, Selenium, Sikuli, ecc. A seconda delle capacità di programmazione del tester. Se il tester non è in grado di scrivere il programma del computer utilizzando diversi linguaggi di programmazione, può utilizzare l'approccio di registrazione e riproduzione fornito da molti strumenti di automazione del test quali QTP, Selenium IDE ecc. Durante la registrazione e la riproduzione, lo strumento genera il codice stesso come parte degli script di automazione del test. In caso contrario, il tester può utilizzare API come Selenium web driver, Sikuli, ecc. E scrivere gli script di test da solo nei diversi linguaggi di programmazione come Java, Ruby, Groovy, PHP, Python, ecc. Tali script di test possono automatizzare gli scenari di test per testare gli elementi grafici che sono presenti sullo schermo sotto test.

- **Test basati su modelli:** una descrizione grafica del comportamento del sistema è nota come modello. Un modello ci aiuta a determinare il comportamento del sistema sotto test. Utilizziamo i requisiti di sistema per generare casi di test efficienti con l'aiuto di un modello. Di seguito è riportata una panoramica di un test basato sul modello.

Capitolo 1 Testing

Il test del software è un'indagine condotta per fornire alle parti interessate informazioni sulla qualità del prodotto software o del servizio sottoposto a test. I test del software possono anche fornire una visione obiettiva e indipendente del software per consentire all'azienda di apprezzare e comprendere i rischi legati all'implementazione del software. Le tecniche di test comprendono il processo di esecuzione di un programma o un'applicazione con l'intento di individuare bug del software (errori o altri difetti) e verificare che il prodotto software sia idoneo all'uso.
Poiché il numero di test possibili per componenti software anche semplici è praticamente infinito, tutti i test del software utilizzano alcune strategie per selezionare test fattibili per il tempo e le risorse disponibili. Di conseguenza, i test del software tipicamente (ma non esclusivamente) tentano di eseguire un programma o un'applicazione con l'intento di trovare bug software (errori o altri difetti). Il lavoro di testing è un processo iterativo come quando un bug è corretto, può illuminare altri bug più profondi o persino crearne di nuovi.

1.1 Testing unitario

l test unitario si riferisce a test che verificano la funzionalità di una sezione specifica di codice, solitamente a livello di funzione. In un ambiente orientato agli oggetti, questo di solito è a livello di classe e i test unitari minimi includono costruttori e distruttori. [44]

Questi tipi di test vengono solitamente scritti dagli sviluppatori mentre lavorano sul codice (stile white-box), per garantire che la funzione specifica funzioni come previsto. Una funzione potrebbe avere più test, per rilevare casi d'angolo o altri rami nel codice. I test unitari da soli non sono in grado di verificare la funzionalità di un pezzo di software, ma piuttosto sono usati per garantire che gli elementi costitutivi del software funzionino indipendentemente l'uno dall'altro.

Il test unitario è un processo di sviluppo software che prevede un'applicazione sincronizzata di un ampio spettro di strategie di prevenzione e rilevamento dei difetti al fine di ridurre i rischi, i tempi ei costi di sviluppo del software. Viene eseguito dallo sviluppatore del software o dall'ingegnere durante la fase di costruzione del ciclo di vita dello sviluppo del software. Il test unitario ha lo scopo di eliminare gli errori di costruzione prima che il codice venga promosso a test aggiuntivi; questa strategia ha lo scopo di aumentare la qualità del software risultante e l'efficienza del processo di sviluppo complessivo.

A seconda delle aspettative dell'organizzazione per lo sviluppo software, test di unità potrebbe includere l'analisi statica del codice , l'analisi del flusso di dati , analisi metriche, le revisioni del codice tra pari, copertura del codice di analisi e di altre pratiche di test del software.

L'unità per definizione è la parte più piccola del software che si intende testare.

Unità possono essere:
• Le funzioni
• I metodi
• Le classi
• I package (note le loro interfacce)
• Gli interi sistemi

Il testing a livello di unità dei comportamenti di una classe dovrebbe essere progettato ed eseguito dallo sviluppatore della classe, contemporaneamente allo sviluppo stesso della classe

Vantaggi:

- Lo sviluppatore conosce esattamente le responsabilità della classe che ha sviluppato e i risultati che da essa si attende ;
- Lo sviluppatore conosce esattamente come si accede alla classe, ad esempio:

    • Quali precondizioni devono essere poste prima di poter eseguire un caso di test;
    • Quali postcondizioni sui valori dello stato degli oggetti devono verificarsi

Svantaggi:

- Lo sviluppatore tende a difendere il suo lavoro … troverà meno errori di quanto possa fare un tester!

Se la progettazione dei casi di test é un lavoro duro e difficile, l'esecuzione dei casi di test é un lavoro noioso e gramo!

L'automatizzazione dell'esecuzione dei casi di test porta innumerevoli vantaggi:

- Tempo risparmiato (nell'esecuzione dei test)
- Affidabilità dei test (non c'é rischio di errore umano nell'esecuzione dei test)
- Riuso (parziale) dei test a seguito di modifiche nella classe

Una soluzione all'automatizzazione del testing è proposta da un membro della famiglia xUnit: JUnit (Java).

JUnit è un semplice framework open source per scrivere ed eseguire test ripetibili. È un'istanza dell'architettura xUnit per i framework di test delle unità.
Le funzionalità di JUnit includono:

- Asserzioni per testare i risultati attesi
- Test di dispositivi per la condivisione di dati di test comuni
- Test runners per i run test

JUnit è stato originariamente scritto da Erich Gamma e Kent Beck.

Capitolo 2 Applicazione

In questo capitolo si illustra la realizzazione di un'applicazione Eclipse in Java che include una GUI per l'interazione *utente-sistema*.

Tale applicazione simula l'uso di uno sportello bancomat tramite il quale l'utente può effettuare operazione di prelievo e deposito di un determinato importo, ed eventualmente di prestito.

Una classe *ContoBancario* contiene le variabili ed i metodi per la simulazione di un conto bancario.
Una classe *Round* invece, sarà atta a contenere un unico metodo definito *static* per l'approssimazione di un dato valore double, in un formato conforme con quello dell'euro (00,00€).

Si riporta quindi il codice delle due classi:

```
package contoBancario;

public class ContoBancario {
    private double saldo;
    private String ID_Conto;
    public ContoBancario() {
        ID_Conto="CB0000";
        saldo=100.00;
    }
    public double getSaldo() {
        return saldo;
    }
    public void setSaldo(double saldo) {
        this.saldo = saldo;
    }
    public String getID_Conto() {
        return ID_Conto;
    }
    public void setID_Conto(String iD_Conto) {
        ID_Conto = iD_Conto;
    }

    //Si suppone un importo intero per un deposito
    public double deposita(int importo){
        this.saldo+=importo;
        return(Round.round(this.saldo));
    }

    public double prelievo(double importo){
        this.saldo-=importo;
        return(Round.round(this.saldo));
    }
}
```

Figura 1 : Codice classe ContoBancario

```java
package contoBancario;

public class Round {
    public Round() {}
    public static double round(double input){
        double output=((int)(input*100));
        output=output/100;
        return(output);
    }
    public static void main(String [] args){
        double rand;
        for(int i=0;i<10;i++){
        rand=Math.random()*100;
        System.out.println("Rand: "+rand);
        System.out.println("Rand troncato : "+Round.round(rand));
        }
    }
}
```

Figura 3 : Codice classe Round

Tramite il plug-in offerto da Eclipse , WindowBuilder(WB) , è stata creata una Application Windo, denominata *ContoBancarioGUI*, che definisce un'interfaccia(GUI) che l'utente può utilizzare per usufruire dei servizi offerti dal sistema.

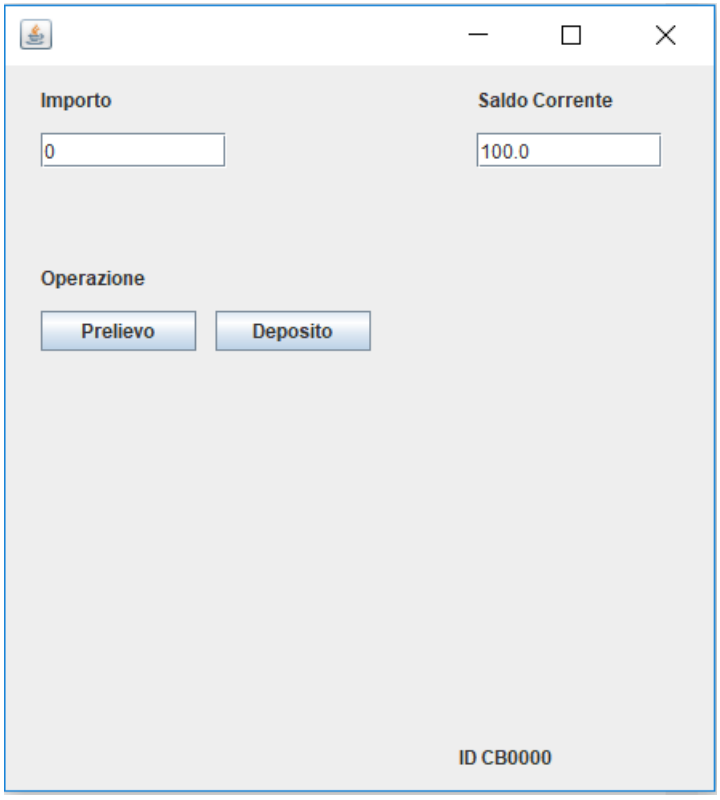

Figura 4 : GUI

Tale interfaccia è stata strutturata seguendo una logica che prevede l'uso di ulteriori componenti nel caso in cui un eventuale prelievo (di un dato importo) porti il saldo corrente in negativo.

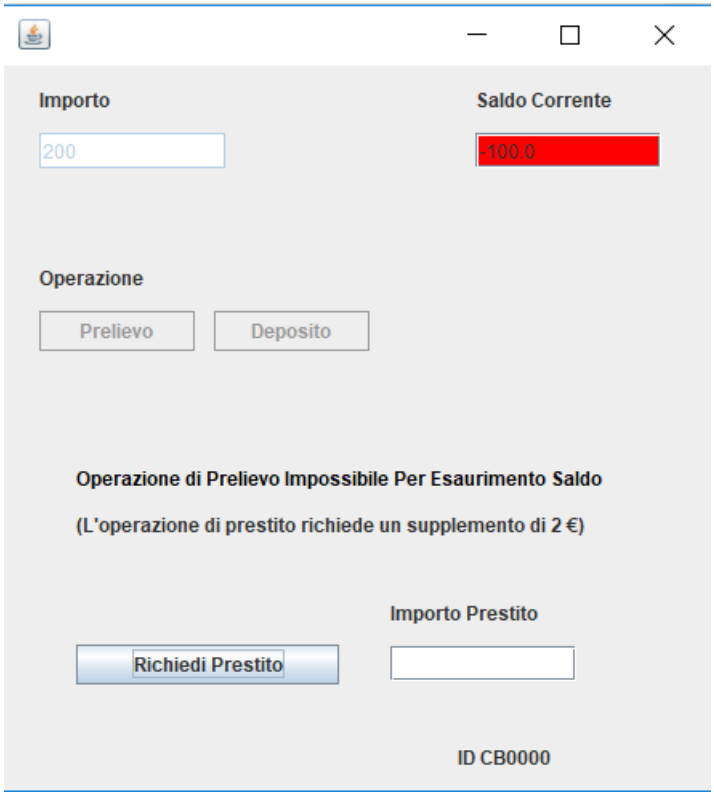

Figura 5 : GUI Saldo negativo

Capitolo 3 JUnit3

Analizziamo i componenti di un test JUnit.

Classi di test: Una classe di test JUnit deve estendere (ereditare da) una classe denominata TestCase della libreria.

Essa deve contenere :
Un metodo *setup()* che viene eseguito prima dell'esecuzione di ogni test

- Utile per eseguire operazioni preliminari necessarie per poter soddisfare le precondizioni comuni a più di un caso di test.

Un metodo *teardown ()* che viene eseguito dopo ogni caso di test

- Utile per eseguire operazioni finali necessarie per poter soddisfare le precondizioni comuni a più di un caso di test

Metodi di test : Ogni metodo di test deve avere un nome che inizia con la parola test (in minuscolo). Possono essere realizzati un numero qualsiasi di metodi di test.

E' fondamentale rimettere sempre l'applicazione e le sue risorse nello stato di partenza al termine di ogni test poiché JUnit non garantisce mai in che ordine verranno eseguiti i test.

3.1 Funzionamento

All'interno di JUnit esiste almeno un metodo *public static*, eseguibile da linea di comando. All'avvio di JUnit, questo metodo :

1 Interroga il programma in cerca di classi di test e metodi in esse (Le librerie di reflection presenti nel linguaggio Java rendono possibili queste interrogazioni);
2 Esegue ciclicamente il metodo *setup()* , poi il metodo test in questione , ed infine il metodo *teardown().*

E' importante notare che durante ognuna di queste esecuzioni , JUnit produce un output separato con l 'esito del test.

La rasformazione di un test in codice JUnit avviene mediante i seguenti passi :

- **Precondizioni**: Tramite il metodo setup vengono eseguite delle operazioni mirante a far diventare vere le precondizioni. Al termine del metodo vengono poste delle asserzioni che valutano le precondizioni: se non fossero verificate, i test non vengono eseguiti;
- **Input** :Tramite settaggi diretti di attributi oppure chiamate a metodi set (che si suppongono corretti e senza necessità di essere testati) ;
- **Test** : Esecuzione del metodo da testare con gli eventuali ulteriori parametri di input relativi a quel caso di test ;
- **Output** : Controllo di espressioni booleane relative alla coincidenza dei valori di output ottenuti con quelli osservati (asserzioni) ;
- **Postcondizioni** : Valutazione di espressioni booleane (asserzioni) relative alle postcondizioni.

3.2 Esempio Applicativo JUnit3

Generiamo una classe di tipo *JUnit Test Case*, che denomineremo *ContoBancarioGUITest*, la quale avrà i classici metodi *setup()* e *teardown()* , oltre ovviamente i metodi per il testing delle funzioni della classe di interesse , nel nostro caso, *ContoBancarioGUI*.

```
public class ContoBancarioGUITest extends TestCase {
    private ContoBancarioGUI c;

    protected void setUp() throws Exception {
        super.setUp();
        c=new ContoBancarioGUI();
        assertNotNull(c);
    }

    protected void tearDown() throws Exception {
        super.tearDown();
        c=null;
        assertNull(c);
    }
```

Figura 6 : setup() e teardown()

```
public void testOnclick_deposita(){
    int importo=100;
    String imp;
    Double new_saldo;

    c.getTextField_Importo().setText(Double.toString(importo));
    imp=c.getTextField_Importo().getText();
    assertEquals("Importo non corretto", imp, Double.toString(importo));

    new_saldo = c.getC().deposita(importo);
    c.getTextField_SaldoCorrente().setText(Double.toString(new_saldo));
    assertEquals("Deposito non corretto", new_saldo , c.getC().getSaldo());
}
```

Figura 7 : testOnclick_deposita()

```
public void testOnclick_preleva(){
    Double importo=99.5;
    String imp;
    Double new_saldo;

    c.getTextField_Importo().setText(Double.toString(importo));
    imp=c.getTextField_Importo().getText();
    assertEquals("Importo non corretto", imp, Double.toString(importo));


    new_saldo = c.getC().prelievo(importo);
    c.getTextField_SaldoCorrente().setText(Double.toString(new_saldo));
    assertEquals("Prelievo non corretto",new_saldo,c.getC().getSaldo());
}
```

Figura 8 : testOnclick_preleva()

```
public void testOnclick_richiediPrestito(){
    Double importo=110.0;
    Double prestito=10.0;
    Double new_saldo;
    String imp;

    c.getTextField_Importo().setText(Double.toString(importo));
    imp=c.getTextField_Importo().getText();
    assertEquals("Importo non corretto", imp, Double.toString(importo));

    if(c.getC().prelievo(importo)<0.0){
        c.getTextFieldRichiediPrestito().setText(Double.toString(prestito));
        prestito=Double.parseDouble(c.getTextFieldRichiediPrestito().getText());
        new_saldo=c.getC().getSaldo()+prestito;
        c.getTextField_SaldoCorrente().setText(Double.toString(new_saldo));
        assertEquals("Richiesta prestito non corretta",new_saldo,0.0);
    }
}
```

Figura 9 : testOnclick_richiediPrestito()

Un asserzione è un'affermazione che può essere vera , o falsa.
I risultati attesi sono documentati con delle asserzioni esplicite, non con delle stampe che comunque richiedono dispendiose ispezioni visuali dei risultati .
Se l'asserzione è :

- vera: il test è andato a buon fine ;
- falsa: il test è fallito ed il codice testato non si comporta come atteso, quindi c'è un errore a tempo dinamico.

Le asserzioni sono utilizzate sia per verificare gli oracoli che le postcondizioni .
Le asserzioni potrebbero essere utilizzate anche per verificare le precondizioni: se la precondizione fallisce, il test non viene proprio eseguito (e viene riportato come fallito).

Il metodo *assertEquals* verifica se il valore corrente nella *TextField_SaldoCorrente* é uguale al valore di *new_saldo* (valore atteso); in caso contrario tale assert restituisce una failure con il messaggio di errore indicato.

Il metodo *asserTrue* verifica la veridicità della condizione esplicitata , ed in caso contrario restituisce una failure con il messaggio specificato.

L'asserzione nel metodo *setUp()* corrisponde alla precondizione: se non è verificata il test non viene eseguito

L'asserzione nel metodo tearDown() corrisponde alla postcondizione: se non è verificata, il test ha rilevato un malfunzionamento

Le asserzioni nei metodi *test* corrispondono all'oracolo: se non sono verificata, il test ha rilevato un malfunzionamento.

Da notare che il test viene eseguito solo se la precondizione è verificata, mentre la postcondizione è testata in ogni caso.

Un'altra caratteristica di JUnit3 è che tutti i metodi per il testing all'interno della classe hanno come prefisso al loro nome la parola "test".
(Es. *testOnClick_preleva*)

Questo è necessario per far si che tali metodi siano eseguiti secondo un processo di test ben definito, che garantisce l'esecuzione di un dispositivo sia prima che dopo il metodo di test.

Eseguendo quindi il file come un JUnit Test , otteniamo i seguenti risultati :

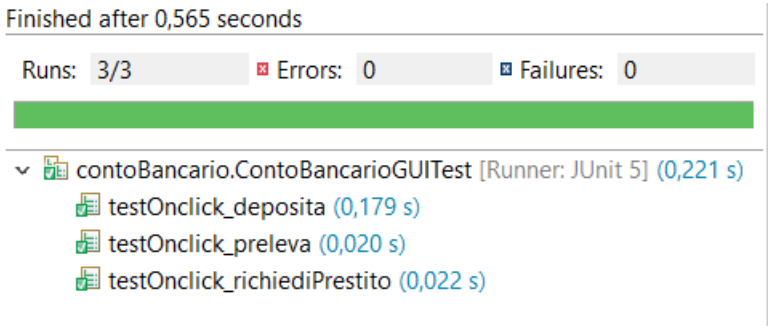

Figura 10 : Risultato test JUnit3

Ciò significa che tutte le asserzioni all'interno dei metodi per il testing sono risultate vere e quindi non si sono verificate delle failure.

Nel caso in cui si fosse verifica una failure, grazie a JUnit possiamo prontamente trovare il rigo con l'asserzione errata e avviare il debugging … Quando pensiamo di aver corretto l'errore rieseguiamo il test.

E’ Buona prassi raggruppare tutti i test cases in test suite , poiché in futuro ulteriori classi di test possono essere aggiunte in seguito modificando il codice generato

```
public class AllTests extends TestCase {

    public static Test suite() {
        TestSuite suite = new TestSuite(AllTests.class.getName());
        //$JUnit-BEGIN$
        suite.addTestSuite(ContoBancarioGUITest.class);
        //$JUnit-END$
        return suite;
    }

}
```

Figura 11 : Test Suite JUnit3

3.3 Limiti di JUnit3

Un metodo di test Junit che si trova in un package di test non può:

- Accedere a metodi e attributi privati della classe testata (Sarebbe dovuto essere nella stessa classe da testare) ;
- Accedere a metodi e attributi protected della classe testata (Dovrebbe ereditare dalla classe da testare ma eredita già da TestCase e l’ereditarietà multipla non è consentita);
- Accedere a metodi e attributi con visibilità package(Sarebbe dovuto essere nello stesso package dell’attributo/metodo da testare).

Esso può Accedere a metodi e attributi pubblici , a patto che il metodo di test non sia nello stesso progetto dell’attributo/metodo .
A tal proposito JUnit 4 è stato introdotto per risolvere la maggior parte di questi problemi e rendere possibile il testing anche ad un livello white box.

Capitolo 4 JUnit4

Grazie alle annotazioni Java 5, JUnit 4 è più leggero e flessibile che mai. Ha abbandonato le sue stringenti convenzioni di denominazione e gerarchie di ereditarietà in favore di alcune nuove funzionalità più Pratiche.
Si introduce inoltre la possibilità di eseguire determinate azioni prima e dopo l'esecuzione di tutti i test di una classe di test ,oppure prima e dopo l'esecuzione di un singolo test.

4.1 JUnit4 vs JUnit3

Prima dell'aggiunta delle annotazioni Java 5 in JUnit 4, il framework aveva stabilito due convenzioni essenziali per la sua capacità di funzionare. Il primo era che JUnit richiedeva implicitamente che qualsiasi metodo scritto per funzionare come test logico inizi con la parola "test" . Qualsiasi metodo che inizi con quella parola verrebbe eseguito secondo un processo di test ben definito che garantiva l'esecuzione di un dispositivo sia prima che dopo il metodo di test. Secondo, per JUnit per riconoscere un oggetto di classe contenente test, la classe stessa doveva estendersi da JUnit TestCase(o qualche sua derivazione). Un test che violasse una di queste due convenzioni non sarebbe stato eseguito .

JUnit 4 utilizza le annotazioni Java 5 per eliminare completamente entrambe le convenzioni. La gerarchia delle classi non è più necessaria e i metodi destinati a funzionare come test devono solo essere decorati con una nuova annotazione: *@Test*.

Rappresentiamo in primis una classe spoglia, al solo scopo di illustrare le prime differenze.

```java
package contoBancario;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class Junit4 {

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void test() {
        fail("Not yet implemented");
    }

}
```

Figura 12 : Test Case JUnit4

@BeforeClass Eseguito una volta, prima dell'inizio di tutti i test. Viene utilizzato per eseguire attività intensive nel tempo, ad esempio per connettersi a un database. I metodi contrassegnati con questa annotazione devono essere definiti static per funzionare con JUnit.

@Before Eseguito prima di ogni test e viene utilizzato per preparare l'ambiente di test (ad esempio, leggere i dati di input, inizializzare la classe).

@After Eseguito dopo ogni test. Viene utilizzato per pulire l'ambiente di test (ad esempio, eliminare i dati temporanei, ripristinare i valori predefiniti). Può anche risparmiare memoria eliminando costose strutture di memoria.

@AfterClass Eseguito una volta, dopo che tutti i test sono stati completati. Viene utilizzato per eseguire attività di pulizia, ad esempio, per disconnettersi da un database. I metodi annotati con questa annotazione devono essere definiti staticper funzionare con JUnit.

@Test è un'annotazione per marcare i metodi che si considerano di Test Non è (più) necessario ma è buona norma usare 'test' come prefisso del nome dei metodi di test (cosa obbligatoria in Junit 3).

La classe non eredita da TestCase ! Questo è un grande vantaggio poiché si potrebbe ereditare dalla classe da testare , per accedere quindi ai suoi metodi *protected*.

4.2 Esempio Applicativo JUnit4

Il notevole vantaggio di JUnit4 è che posso scrivere il codice di testing , ovvero i metodi marcati dall'annotazione @Test , direttamente all'interno della classe da testare.

Definiamo quindi all'interno della classe *ContoBancarioGUI* tutti i metodi per il testing necessari.

In questo caso si è pensato di effettuare , per i valori utili al testing , una lettura da file. Ciò è stato possibile grazie alle classi *BufferedReader* e *FileReader* offerte dalla libreria java.

I file *TestDataDeposita* e *TestDataPreleva* sono stati inseriti nel path contenente il progetto e sono in sostanza dei file testuali contenenti i valori usati per il testing.

```
@BeforeClass
public static void setUpBeforeClass() throws Exception {
    reader_deposita=new BufferedReader(new FileReader("TestDataDeposita.txt"));
    assertNotNull("Fail nel BeforeClass (reader deposita) ",reader_deposita);
    reader_preleva=new BufferedReader(new FileReader("TestDataPreleva.txt"));
    assertNotNull("Fail nel BeforeClass (reader preleva) ",reader_preleva);
}

@AfterClass
public static void tearDownAfterClass() throws Exception {
    reader_deposita=null;
    reader_preleva=null;
    assertNull("Fail nell' AfterClass (reader deposita) ",reader_deposita);
    assertNull("Fail nell' AfterClass (reader preleva) ",reader_preleva);
}

@Before
public void setUp() throws Exception {
    assertNotNull("Fail setUP()",frame);
}


@After
public void tearDown() throws Exception {
    frame=null;
    assertNull("Fail tearDown() ",frame);
}
```

Figura 13 : Metodi Standard JUnit4

```
@Test
public void TestDeposita(){
    int importo;
    String s_saldo_corrente;
    double saldo_corrente;
    double new_saldo;
    while(true){
        try {
            data=reader_deposita.readLine();
        } catch (IOException e1) {
            e1.printStackTrace();
        }
        if(data==null)
            break;

        textField_Importo.setText(data);
        importo=Integer.parseInt(data);
        s_saldo_corrente = textField_SaldoCorrente.getText();
        saldo_corrente=new Double(s_saldo_corrente);
        new_saldo=c.deposita(importo);
        textField_SaldoCorrente.setText(Double.toString(new_saldo));
        assertTrue("Failure in TestDeposita", (new_saldo)==(importo+saldo_corrente));
    }
}
```

Figura 14 : TestDeposita()

```
@Test
public void TestPreleva(){
    double importo;
    String s_saldo_corrente;
    double saldo_corrente;
    double new_saldo;
    while(true){
        try {
            data=reader_preleva.readLine();
        } catch (IOException e1) {
            e1.printStackTrace();
        }
        if(data==null)
            break;

        textField_Importo.setText(data);
        importo=Double.parseDouble(data);
        s_saldo_corrente = textField_SaldoCorrente.getText();
        saldo_corrente= Double.parseDouble(s_saldo_corrente);
        new_saldo=c.prelievo(importo);
        textField_SaldoCorrente.setText(Double.toString(new_saldo));
        assertTrue("Failure in TestPreleva",(new_saldo)==Round.round(saldo_corrente-importo));
    }
}
```

Figura 15 : TestPreleva()

```java
@Test
public void TestRichiestaPrestito(){
        int importo=110;
        int prestito=20;
        int supplemento=2;
        double saldo_corrente;
        textField_Importo.setText(Double.toString(importo));
        saldo_corrente=Double.parseDouble(textField_SaldoCorrente.getText());
        System.out.println("Saldo corrente "+saldo_corrente);
        double new_saldo1=c.prelievo(importo);
        if(new_saldo1<0.0){
            double new_saldo2=c.deposita(prestito-supplemento);
            textField_SaldoCorrente.setText(Double.toString(new_saldo2));
            assertTrue("Failure ",new_saldo2==8);
        }
}
```

Figura 16 : TestRichiestaPrestito()

L'assert qui utilizzata è assertTrue(String message, boolean condition) la quale asserisce ad una condizione vera.

Eseguendo quindi il file come un JUnit Test , otteniamo i seguenti risultati :

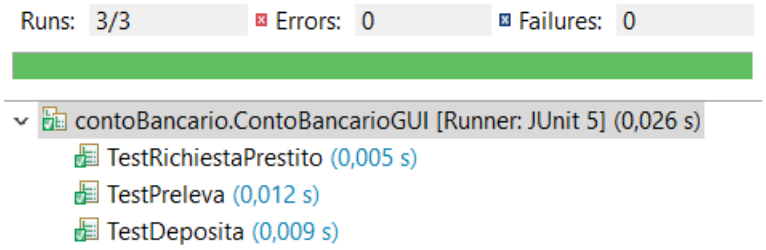

Figura 17 : Risultati test JUnit4

Come per JUnit3 , non avendo rilevato alcune asserzione falsa , l'esito è positivo per tutti e tre i metodi.

Anche in JUnit 4 è usato il Test Suite , ovvero un meccanismo che raggruppa logicamente dei test e li esegue assieme.

```
import org.junit.runner.RunWith;

@RunWith(Suite.class)
@SuiteClasses({ ContoBancarioGUI.class })
public class AllTests {

}
```

Figura 18 : Test Suite JUnit4

L'annotazione *@SuiteClasses* raggruppa una lista di classi, passate come parametro, in una test suite , mentre l'annotazione *@RunWith* permette di impostare diversi esecutori di test.

4.3 Limiti di JUnit4

JUnit 4 presenta alcune chiare limitazioni:

- L'intero framework è contenuto in un'unica libreria jar.
  Quest'ultima deve essere importata anche quando è richiesta solo una particolare funzione.
  In JUnit 5, otteniamo più granularità e possiamo importare solo ciò che è necessario;

- Un test runner può eseguire solo un test alla volta in JUnit.
  JUnit 5 consente a più runners di lavorare contemporaneamente;

- JUnit 4 non è mai avanzato oltre Java 7, perdendo molte funzionalità di Java 8.
  JUnit 5 fa buon uso delle funzionalità di Java 8;

L'idea alla base di JUnit 5 era di riscrivere completamente JUnit 4 per risolvere la maggior parte di questi inconvenienti.

Capitolo 5 Maveryx

Maveryx è un tool open source per il test automatico di tipo funzionale e di regressione per applicazioni scritte in Java. Esso fornisce al tester un insieme di librerie che permettono di eseguire azioni su elementi grafici del sistema da collaudare. Maveryx elimina completamente la dipendenza dal GUI map. L'interfaccia utente viene mappata direttamente al runtime e gli elementi da testare vengono identificati mediante l'Intelligent GUI Objects Finder che utilizza alcuni algoritmi di ricerca per fornire tale funzionalità. Maveryx utilizza algoritmi di localizzazione di tipo fuzzy in caso che le informazioni fornite nei test script siano parziali o nel caso in cui l'elemento da testare sia cambiato. Maveryx, quindi, permette di ridurre significativamente i tempi e i costi nel creare e manutenere i test script.

5.1 Eliminazione dei GUI Maps

Una delle grandi sfide nel test automatico è quello di dover interagire con le interfacce grafiche. Molti tool per il test automatico utilizzano il "GUI maps" per riconoscere e localizzare gli oggetti da verificare.
Una "GUI map" è una vista statica che descrive l'applicazione sotto test. Gli strumenti che ricorrono ad essa, leggono le descrizioni fornite dalla mappa per trovare gli elementi, collocati nell'interfaccia grafica dell'applicazione da testare, che abbiano le caratteristiche descritte.
In tal modo ,quando l'interfaccia utente viene modificata ,si rende necessario aggiornare i "GUI maps" affinché riflettano tali modifiche.
Diversamente da altri strumenti, Maveryx analizza ed individua gli oggetti da testare e le loro caratteristiche, valutando tutto al runtime durante l'esecuzione dello script, senza utilizzare alcuna mappa grafica. Quando un test viene eseguito Maveryx effettua degli "snapshots" dell'interfaccia utente dell'applicazione da collaudare. Ogni "snapshot" viene elaborato dall' *Intelligent GUI Objects Finder.* Quest'ultimo, mediante l'utilizzo di particolari algoritmi di ricerca, riesce ad identificare e a localizzare gli elementi dell'interfaccia grafica su cui vanno effettuati le operazioni richieste. Maveryx analizza ed identifica un oggetto nello stesso modo con cui una persona, guardando un'immagine individua un punto ben preciso nel quale è collocato l'elemento di interesse, in questo modo si possono sviluppare automaticamente i test script prima che l'applicazione venga rilasciata e pronta per il testing.

5.2 Esempio Applicativo

Capitolo 6 JUnit5

A differenza delle versioni precedenti di JUnit, JUnit 5 è composto da tre diversi sotto-progetti.

JUnit 5 = Piattaforma JUnit + JUnit Jupiter + JUnit Vintage

La piattaforma JUnit funge da base per l'avvio di framework di test su JVM.
Definisce inoltre l'API TestEngine per lo sviluppo di un framework di test che viene eseguito sulla piattaforma. Inoltre, la piattaforma fornisce un Launcher della console per avviare la piattaforma dalla riga di comando e creare plugin per Gradle e Maven e un Runner basato su JUnit 4 per eseguire qualsiasi TestEngine sulla piattaforma.

JUnit Jupiter è la combinazione del nuovo modello di programmazione e del modello di estensione per la scrittura di test ed estensioni in JUnit 5. Il sottoprogetto Jupiter fornisce un TestEngine per eseguire test basati su Jupiter sulla piattaforma.

JUnit Vintage fornisce un TestEngine per eseguire test basati su JUnit 3 e JUnit 4 sulla piattaforma.

JUnit 5 richiede Java 8 (o successivo) in fase di esecuzione. Tuttavia, è ancora possibile testare il codice che è stato compilato con le versioni precedenti di JDK.

6.1 JUnit5 VS JUnit4

JUnit 4 presenta alcune chiare limitazioni:

- L'intero framework era contenuto in un'unica libreria jar. L'intera libreria deve essere importata anche quando è richiesta solo una particolare funzione. In JUnit 5, otteniamo più granularità e possiamo importare solo ciò che è necessario;
- Un test runner può eseguire solo test alla volta in JUnit 4. JUnit 5 consente a più runners di lavorare contemporaneamente
- JUnit 4 non è mai avanzato oltre Java 7, perdendo molte funzionalità di Java 8.

Molte annotazioni sono state aggiunte/modificate in JUnit5 come ad esempio :

@Test Denota che un metodo è un metodo di prova. A differenza dell'annotazione *@Test* di JUnit 4, questa annotazione non dichiara alcun attributo, poiché le estensioni di test in JUnit Jupiter operano in base alle proprie annotazioni dedicate. Tali metodi vengono ereditati a meno che non vengano sovrascritti.

@ParameterizedTest Indica che un metodo è un test parametrizzato. Tali metodi vengono ereditati a meno che non vengano sovrascritti.

@RepeatedTest Indica che un metodo è un modello di test per un test ripetuto. Tali metodi vengono ereditati a meno che non vengano sovrascritti.

@TestFactory Denota che un metodo è una fabbrica di test per i test dinamici. Tali metodi vengono ereditati a meno che non vengano sovrascritti.

@TestInstance Utilizzato per configurare il ciclo di vita dell'istanza di test per la classe di test annotata. Tali annotazioni sono ereditate.

@TestTemplate Indica che un metodo è un modello per i casi di test progettati per essere invocati più volte a seconda del numero di contesti di invocazione restituiti dai provider registrati. Tali metodi vengono ereditati a meno che non vengano sovrascritti.

@DisplayName Dichiara un nome di visualizzazione personalizzato per la classe di test o il metodo di prova. Tali annotazioni non sono ereditate.

Altre annotazioni che sono state modificate all'interno di JUnit 5:

@Before viene rinominata in *@BeforeEach*

@After viene rinominata in *@AfterEach*

@BeforeClass viene rinominata in *@BeforeAll*

@AfterClass viene rinominata in *@AfterAll*

@Ignore viene rinominata in *@Disabled*

6.2 Esempio Applicativo

Definiamo all'interno della classe *ContoBancarioGUI* tutti i metodi per il testing necessari.

```
@BeforeEach
void setUp() throws Exception {
    assertNotNull("Fail setUP()",frame);
}

@AfterEach
void tearDown() throws Exception {
    frame=null;
    assertNull("Fail tearDown() ",frame);
}
```

Non è vista necessaria la presenza di *@BeforeAll* ed *@AfterAll.*

A differenza dell'esempio applicativo visto nel paragrafo 4.2 , vediamo adesso la presenza dei test parametrizzati.

I test parametrizzati consentono di eseguire un test più volte con argomenti diversi. Vengono dichiarati come normali metodi @Test, ma utilizzano invece l'annotazione @ParameterizedTest. Inoltre, è necessario dichiarare almeno una fonte che fornirà gli argomenti per ogni chiamata.
Come fonte per gli argomenti si è optato per l'uso di @CsvFileSource consente di utilizzare file CSV dal classpath. Ogni riga da un file CSV risulta in una chiamata del test parametrizzato.
Il file CVS è nel formato "valori separati con la virgola".
E' possibile quindi passare più parametri al test , coerentemente con i valori separati da una virgola all'interno del CSV.

```
@DisplayName("Test per la funzionalità di deposito")
@ParameterizedTest
@CsvFileSource(resources = "/TestDataDeposita.csv")
void TestOnDeposita(int importo) {
    double saldo_corrente,new_saldo;
    textField_Importo.setText(Integer.toString(importo));
    saldo_corrente=Double.parseDouble(textField_SaldoCorrente.getText());
    new_saldo=saldo_corrente+importo;
    assertTrue("Failure in preleva ",new_saldo==c.deposita(importo));
}
```

Figura 19 : TestOnDeposita

```java
@DisplayName("Test per la funzionalità di prelievo")
@ParameterizedTest
@CsvFileSource(resources = "/TestDataPreleva.csv")
void TestOnPreleva(double importo) {
    double saldo_corrente,new_saldo;
    textField_Importo.setText(Double.toString(importo));
    saldo_corrente=Double.parseDouble(textField_SaldoCorrente.getText());
    new_saldo=Round.round(saldo_corrente-importo);
    assertTrue("Failure in preleva ",new_saldo==c.prelievo(importo));
}
```

Figura 20 : TestOnPreleva

Eseguendo quindi il file come un JUnit Test , otteniamo i seguenti risultati :

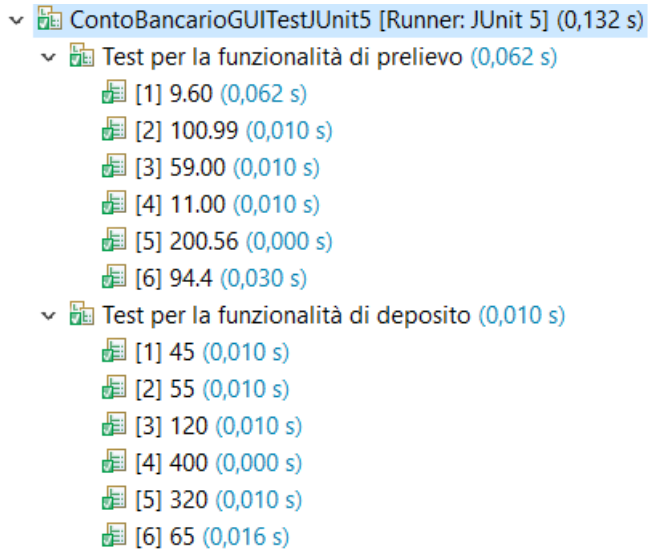

Figura 21 : Risultati test JUnit5

A questo punto si è entrati nell'ottica dei Nested Test.
I test annidati offrono allo scrittore dei test più capacità di esprimere la relazione tra diversi gruppi di test.
Solo le classi annidate non statiche (cioè le classi interne) possono essere utilizzate come classi di test @Nested. L'annidamento può essere arbitrariamente profondo e tali classi interne sono considerate membri a pieno titolo della famiglia della classe di test con un'eccezione: i metodi @BeforeAll e @AfterAll non funzionano per impostazione predefinita. Il motivo è che Java non consente i membri statici nelle classi interne.
Tuttavia, questa limitazione può essere aggirata annotando una classe di test @Nested con @TestInstance

Andiamo quindi ad illustrare il codice di 2 classi Nested Annidate:

```
private double saldo_corrente;
@Nested
@DisplayName("Prelievo")
class Prelievo{
    @BeforeEach
    void setUp() throws Exception {
        saldo_corrente=Double.parseDouble(textField_SaldoCorrente.getText());
    }
    @AfterEach
    void tearDown() throws Exception {
    }

    @Test
    @DisplayName("Test Prelievo")
    void prelievo() {
        double saldo_corrente=Double.parseDouble(textField_SaldoCorrente.getText());
        double new_saldo_corrente=c.prelievo(200);
        assertTrue(new_saldo_corrente!=Round.round(saldo_corrente-200));
    }
```

Figura 22 : 1° Nested Class

```java
@Nested
@DisplayName("Dopo prelievo")
class SaldoNegativo{
    @BeforeEach
    void setUp() throws Exception {
        saldo_corrente=saldo_corrente-200;
        assertTrue(saldo_corrente<0.0,"Saldo Positivo");
    }

    @AfterEach
    void tearDown() throws Exception {
    }

    @ParameterizedTest
    @ValueSource(ints = 200)
    @DisplayName("Test per la funzionalità di richiesta prestito")
    void testOnRichiediPrestito(int importo) {
        double new_saldo_beforePrestito,new_saldo_afterPrestito;
        //supponiamo una qta fissa per il prestito
        int prestito=100;
        saldo_corrente=Double.parseDouble(textField_SaldoCorrente.getText());
        new_saldo_beforePrestito=Round.round(saldo_corrente-importo);
        if(new_saldo_beforePrestito<=0.0) {
            new_saldo_afterPrestito=Round.round(new_saldo_beforePrestito+prestito);
            assertAll(() -> assertTrue(new_saldo_beforePrestito==c.prelievo(importo),"Fail prelievo" ),
                      () -> assertTrue(new_saldo_afterPrestito==c.deposita(prestito),"Fail Deposita "));
        }
    }

}
```

Figura 23 : 2° Nested Class

Il notevole vantaggio è che possiamo eliminare la duplicazione di codice .
Questo poiché le istruzioni presenti nei metodi di setUp() delle Nested Class saranno eseguiti seguendo la gerarchia di annidamento , e dunque è possibile evitare di riiscrivere il codice di tali metodi per ogni test da eseguire.

Conclusioni

Bibliografia

[1] Porfirio Tramontana, Slides Ingegneria Del software II

[2] JUnit5 , https://junit.org/junit5/docs/current/user-guide/