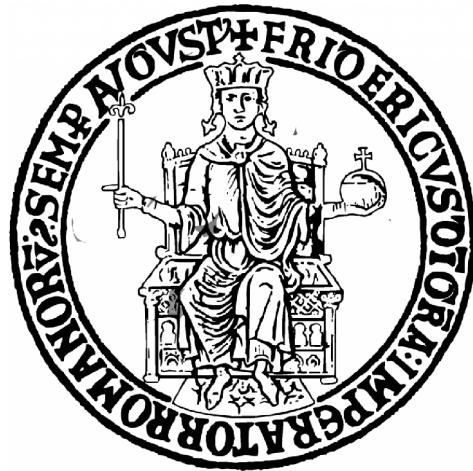


# Università degli Studi di Napoli Federico II

## Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica



## Corso di Ingegneria del Software II

A.A 2013/2014

Prof. Porfirio Tramontana

*Testing - “Ricorda Password”*

Sajmir Rusi            M63/42  
s.rusi@studenti.unina.it

## Sommario

Introduzione .....	.3
User Interface Testing .....	.5
Testing Funzionale .....	13
Testing Brack Box delle Partizioni.....	13
Verifica della copertura White-box .....	18
Copertura Testing della GUI .....	18
Copertura testing Funzionale .....	19
Debugging e Testing di Regressione .....	21
Debugging.....	21
Testing di regressione.....	27
Bibliografia.....	29

# Introduzione

Il presente elaborato ha come obiettivo la verifica e la validazione dell'applicazione "Ricorda Password".

Tale applicazione è priva di documentazione descrittiva e l'elaborazione della stessa è stata fondata principalmente dall'esplorazione, andando a costruire un Finite State Machine (FSM) tramite reverse engineering, a partire dalla UI effettivamente implementata.

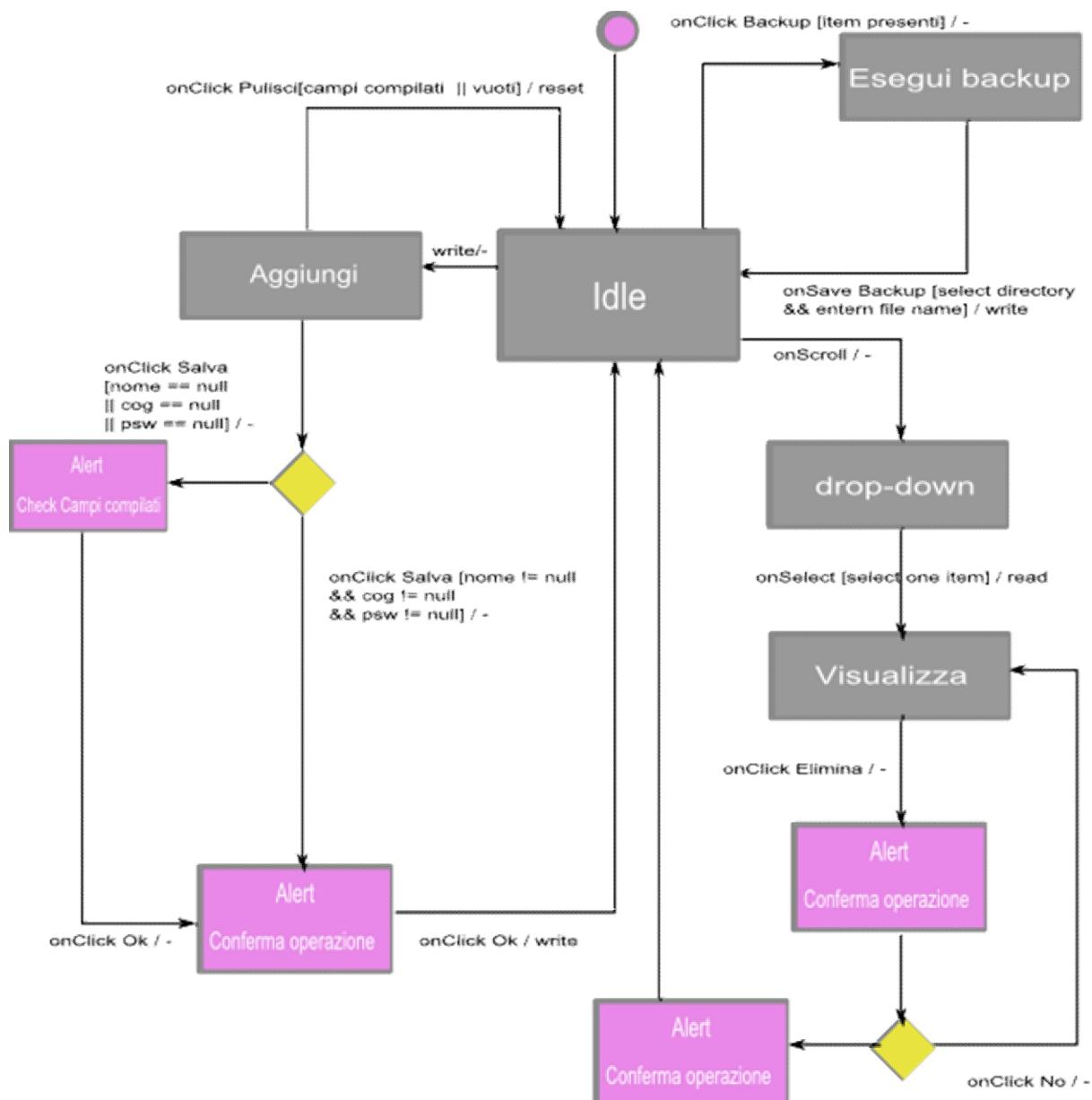


Figura 1 Finite State Machine che descrive la User Interface del applicativo

Il vantaggio è dato sicuramente dall'intuitività della interfaccia grafica e delle funzionalità che ci si aspetta dal applicativo, conseguentemente, utilizzata per la verifica del software.

Tale diagramma **FSM** è caratterizzato da **Stati**: corrispondenti alle interfacce grafiche (o parte di loro) presentabili all'utente e **Transazioni** le quali sono identificate dagli archi e consentono il passaggio da uno stato all'altro.

Per ciascuna transazione si può notare la presenza di tre parametri:

#### **trigger [guardia]/azione**

1. **Trigger** che indica l'evento che attiva il passaggio di stato;
2. **Guardia** è un'espressione predicativa associata ad un evento, che stabilisce una condizione booleana che deve essere verificata affinché scatti la transizione;
3. **Azione** rappresenta il risultato, l'output o l'operazione che segue un evento.

Nel caso in cui sulla transazione sia presente esclusivamente il trigger sarà necessario testare il funzionamento dell'evento che consente il passaggio di stato. Qualora la transizione sia identificata dalla coppia trigger[guardia] allora oltre alla verifica dell'evento, occorrerà verificare anche il valore della guardia stessa.

E' stato denominato stato *Idle*, l'interfaccia principale dell'applicativo in assenza di dati visualizzati o inseriti e senza sollecitazione dall'esterno.

# User Interface Testing

A partire dalla FSM, costruita precedentemente, è stato preso in considerazione la scelta di effettuare un testing della GUI secondo il criterio di copertura delle transizioni. Tale testing prevede che ogni transizione sia percorsa almeno una volta durante l'esecuzione del test suite.

I casi di test che sono stati scelti per massimizzare la copertura sono presentati di seguito:

Test Case	Stato Iniziale	Sequenza di eventi	Stati Intermedi	Stato Finale
RpswGUI1	Idle	-onScrolldown -onSelect	-drop-down-list	Visualizza

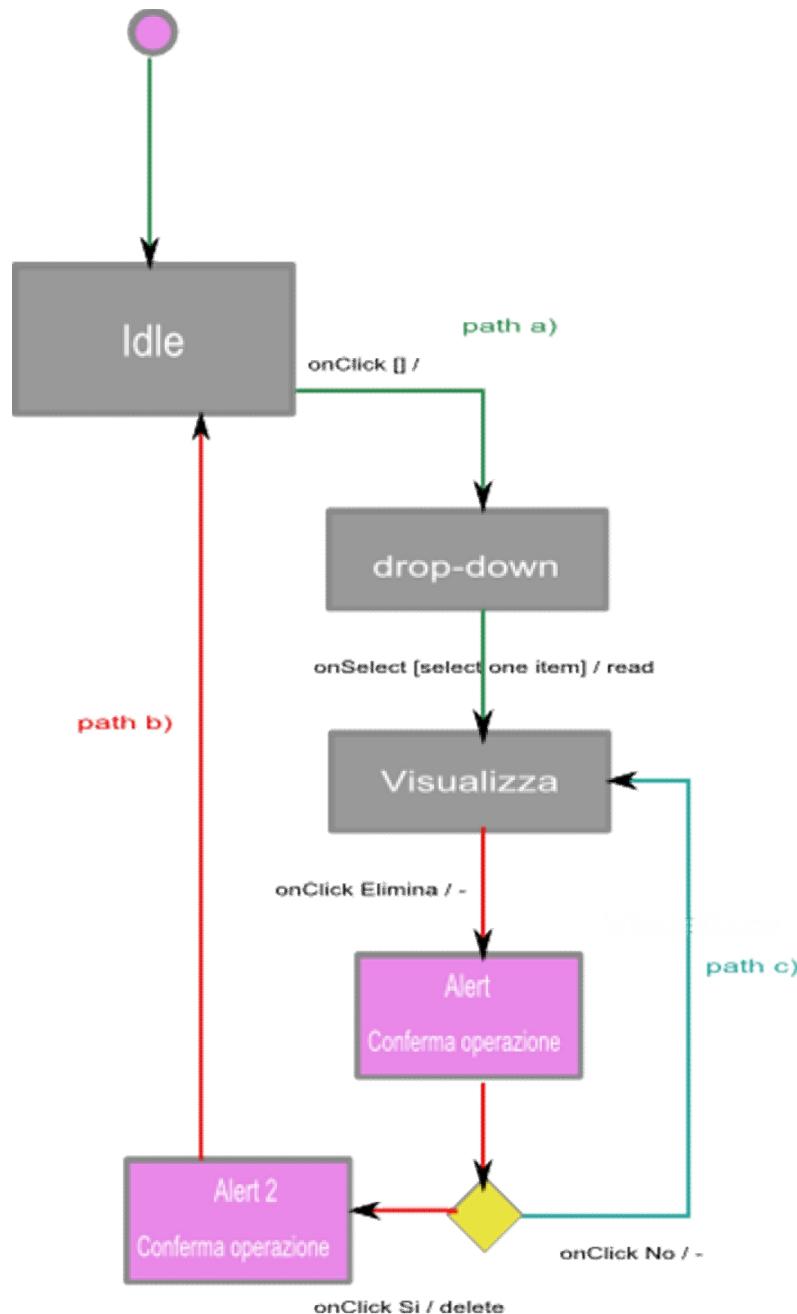
Tabella 1 path (a corrisponde all'evento visualizza un elemento)

Test Case	Stato Iniziale	Pre Condizione	Sequenza di eventi	Stati Intermedi	Stato Finale	Post condizione
RpswGUI2	Idle	-tupla presente nella drop-down list	-onScrolldown -onSelect -onClick Elimina -onClick Si -onClick Si	- drop-down -Visualizza -Alert -Alert 2	Idle	-tupla non presente nella drop-down list

Tabella 2 path b) corrisponde all'evento elimina un element

Test Case	Stato Iniziale	Pre Condizione	Sequenza di eventi	Stati Intermedi	Stato Finale	Post condizione
RpswGUI3	Idle	-tupla presente nella drop-down list	-onScrolldown -onSelect -onClick Elimina -onClick No	- drop-down -Visualizza -Alert conferma	Visualizza	-la tupla rimane presente nella drop-down list

Tabella 3 path c)



**Figura 2 Parte della FSM usate per progettare i test cases RpswGUI1, RpswGUI2 e RpswGUI3**

I primi tre casi di test coprono i path presentati in Figura 2 Parte della FSM usate per progettare i test cases RpswGUI1, RpswGUI2 e RpswGUI3corrispondono alle operazioni di visualizzazione di una tripletta dalla drop-down list e una sua eventuale eliminazione.

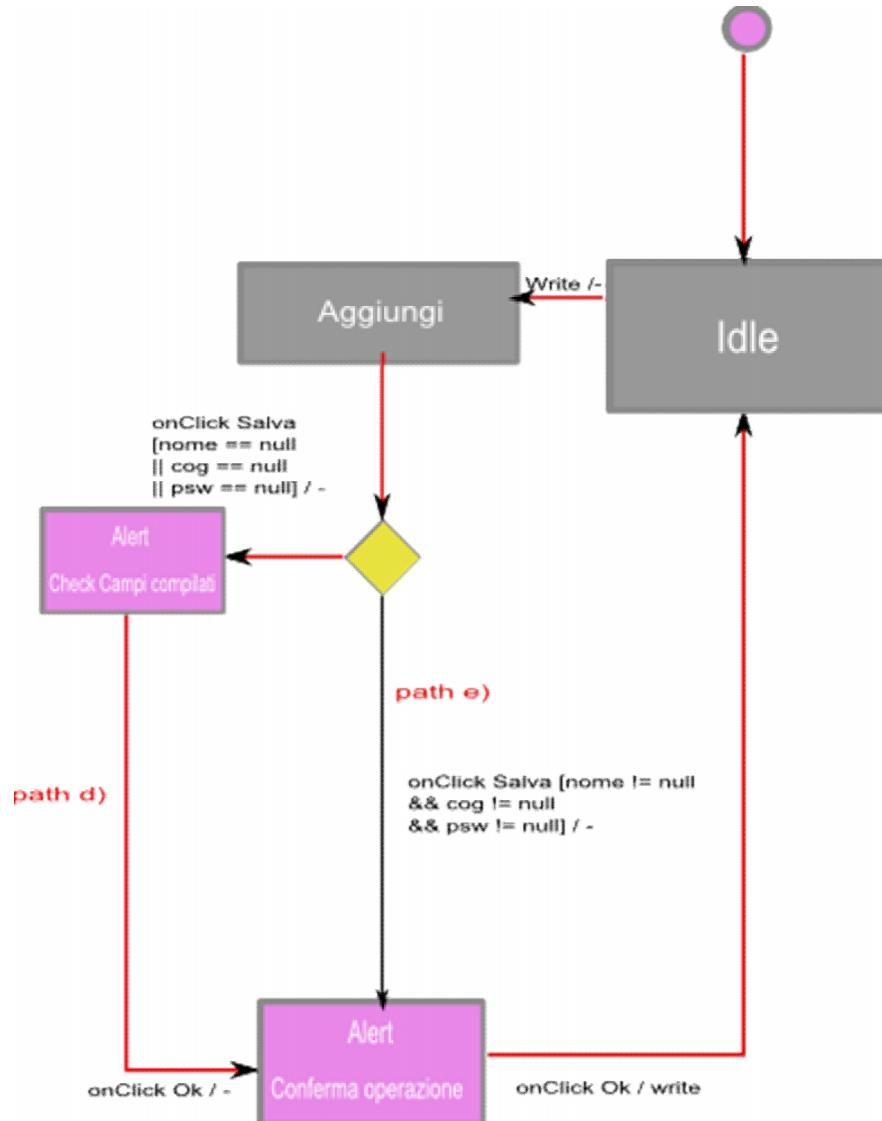
Test Case	Stato Iniziale	Pre Condizione	Sequenza di eventi	Stati Intermedi	Stato Finale	Post Condizione
RpswGUI4	Idle	-n tuple presenti	-onFill_nom -onFill_cog -onFill_psw -onClick Salva -onClick Ok -onClick Ok	-Alert err -Alert succ	-Idle	-n tuple presenti

Tabella 4 path d)

Test Case	Stato Iniziale	Pre Condizione	Sequenza di eventi	Stati Intermedi	Stato Finale	Post Condizione
RpswGUI6	Idle	-N tuple presenti	-onFill_nom -onFill_cog -onFill_psw -onClick Salva -onClict Ok	-Alert successo	Idle	N+1 tuple presenti Valori correttamente inserite

Tabella 5 path e)

Considerando che non si ha informazione sui valori accettabili nei campi di input dato, per questa tecnica di testing si è considerato come classe di equivalenza valida tutto il set di caratteri asci.



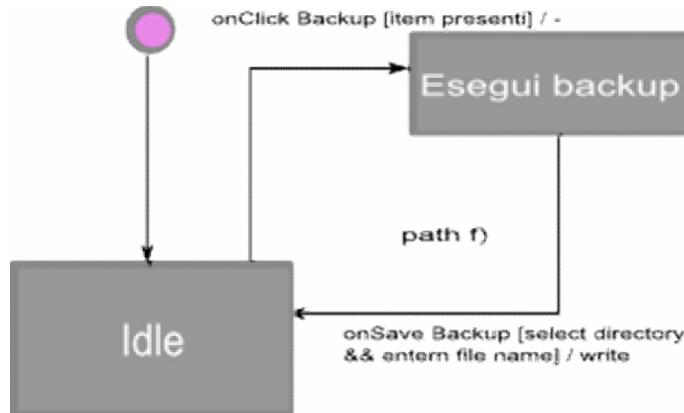
**Figura 3 Parte della FSM usate per progettare i test cases RpswGUI4 e RpswGUI5**

Questi ulteriori casi di test, come si può vedere in Figura 3 Parte della FSM usate per progettare i test cases RpswGUI4 e RpswGUI5vanno a coprire altri due path della FSM e corrispondono alle operazioni di una tripletta con i tre campi compilati correttamente, nonché e di una tripletta con uno o più campi vuoti.

TestCase	Stato Iniziale	Sequenza di eventi	Stati Intermedi	Stato Finale	Post condizioni
RpswGUI7	Idle	-onClick Esegui backup -onSelect dir -onFill name -onClick Salva	waiting	Idle	BackUp Presente nella Sys dir

Tabella 6 path f)

Questo test case mira a ricoprire un path che parzialmente è responsabilità del nostro software. Difatti il programma è responsabile solo dell'avviamento della finestra di resource manager e, in seguito, sarà il sistema operativo responsabile della memorizzazione del file di backUp.



**Figura 4 Parte della FSM usata per progettare il caso di test RpswGUI7**

TestCase	Stato Iniziale	Sequenza di eventi	Stati Intermedi	Stato Finale	Post condizioni
RpswGUI8	Idle	-onFill_nom -onFill_cog -onFill_psw -onClick Pulisci	-----	Idle	Campi: -nom, -cog, -psw vuoti

**Tabella 7 path g)**

L'ultimo path che rimane da coprire è quello in Figura 5 Parte della FSM utilizzata per progettare il test case RpswGUI8che presenta **il funzionamento** di reset dei campi di input dopo una loro compilazione.

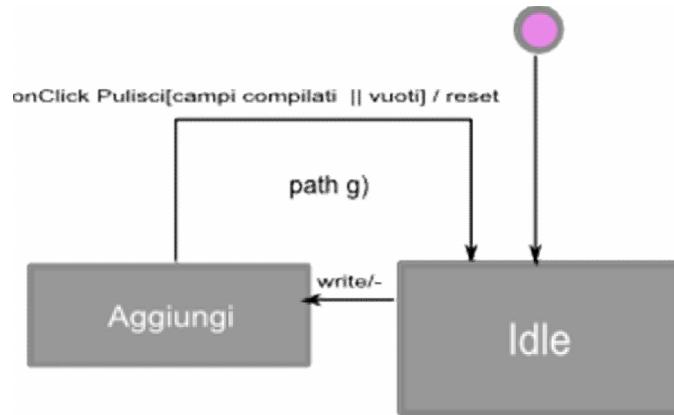


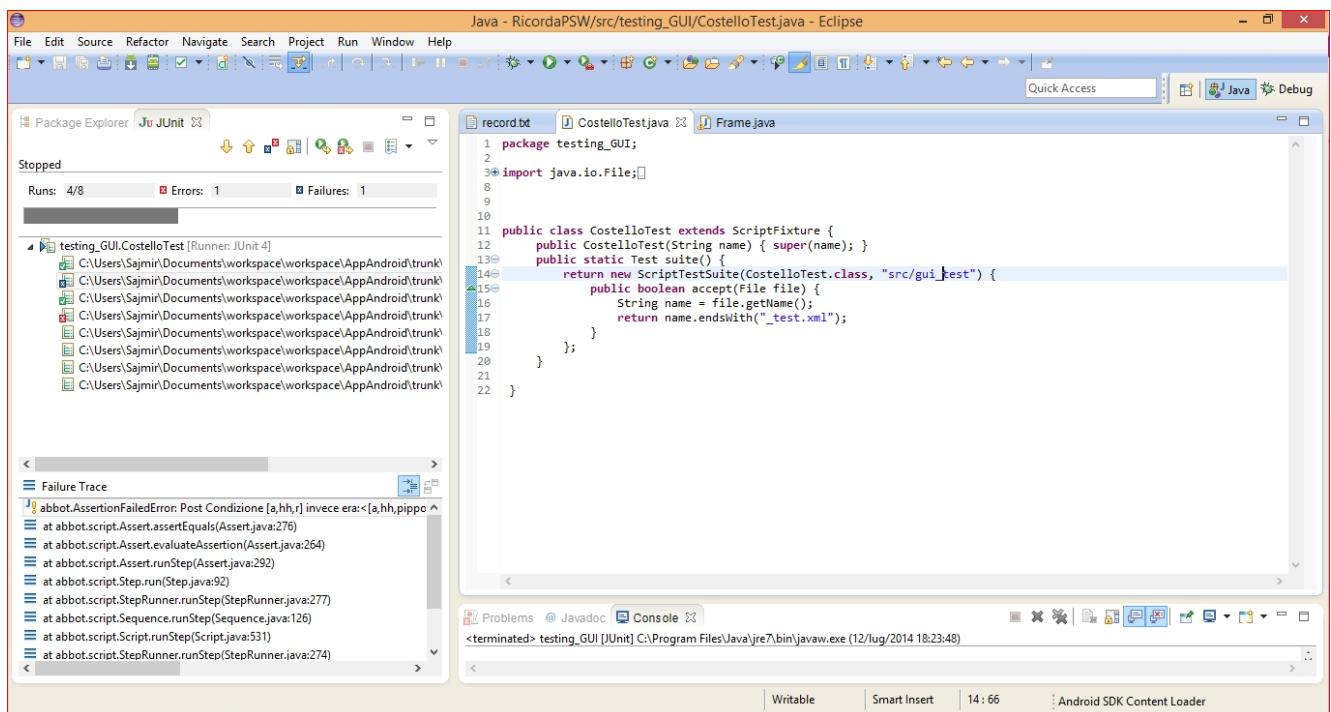
Figura 5 Parte della FSM utilizzata per progettare il test case RpswGUI8

L'esecuzione della test suite ha messo in mostra diversi errori.

La prima failure è scoperta percorrendo il path evidenziato con rosso in Figura 2 Parte della FSM usate per progettare i test cases RpswGUI1, RpswGUI2 e RpswGUI3. Infatti, il secondo test case *RpswGUI2\_test* consente di testare l'eliminazione di un elemento dalla lista delle password salvate. Il successo o meno di tale operazione è validato tramite un'asserzione sul post condizione del test case, che va a indagare la presenza nella drop-down list dell'elemento appena eliminato. Il test case in oggetto ha permesso di scoprire che l'elemento appena eliminato non è stato rimosso dalla lista.

La seconda failure invece riguarda il path evidenziato in rosso in Figura 3 Parte della FSM usate per progettare i test cases RpswGUI4 e RpswGUI5. Con il test case progettato per coprire tale path si era cercato di salvare un item compilando solo due campi. Come si poteva notare anche dalla FMS, il salvataggio avviene in modo incorretto, il passaggio per lo stato *Alert Conferma Operazione* non dovrebbe verificarsi.

Il verificarsi della prima di queste due failure non ci consente di proseguire con le altre test case, poiché non è possibile rispristinare lo stato iniziale dell'applicazione alla fine di ogni test case.



**Figura 6 Failure della GUI test suite**

Va tenuto in considerazione che questa test suite prevede che il programma si trovi in uno stato “così come” è stato rilasciato dal docente, comprendente le diverse password di default salvate.

# Testing Funzionale

## Testing Brack Box delle Partizioni

Come altra tecnica di test si è considerato di usare il testing con Classi di Equivalenza. Per ogni campo d'input sono state considerate delle classi con valori validi e delle classi con valori non validi. Verosimilmente, il programma si dovrebbe comportare nello stesso modo per input appartenente alle classi di equivalenza considerate.

Condizioni sul input: "Nome"

- Il campo "Nome" può assumere qualsiasi valore alfanumerico o di caratteri speciali
- Classi di Equivalenza
  - Valide
    - CE1 "pippo",
    - CE2 "ç°§@#\*+}{[]"
    - CE3 123456
  - Classi non validi
    - CE4 Valore di stringa nulla ""

Condizioni sull'input: "Password"

- Il campo "Password" può assumere qualsiasi valore alfanumerico o di caratteri speciali
- Classi di Equivalenza
  - Valide
    - CE5 "topolino",
    - CE6 "ç°§@\*+}{[]"
    - CE7 1234567
  - Classi non valide
    - CE8 Valore di stringa nulla ""

Condizioni sul input: "Username"

- Il campo "Username" può assumere qualsiasi valore alfanumerico o di caratteri speciali
- Classi di Equivalenza
  - Valide
    - CE9 "paperino",
    - CE10 "ç°§#\*+}{[]"
    - CE11 1234568
  - Classi non validi
    - CE12 Valore di stringa nulla ""

Per generare i casi di test ci basiamo sulla tecnica di copertura delle classi di equivalenza adiacenti, secondo la quale; “*ogni classe di equivalenza deve essere coperta da almeno un caso di test e per ogni caso di test ne esiste almeno un altro che differisce di un solo classe di equivalenza*”.

Avvalendosi del tool online (Tobias : Fourier) abbiamo generato tutte le combinazioni di input possibili considerando le classi di equivalenza individuate. Dai casi di test ottenuti sono stati scelti solo quelli che ci consentono di avere una copertura delle classi di equivalenza adiacenti. Inoltre è stato aggiunto un caso di test che considera la possibilità di avere tutti gli input nulli. Infine, si è stati attenti nella modifica del valore di input “Nome” dai casi di test ottenuti rendendolo univoco per aver modo di reperirlo dall'applicazione. E' stato aggiunto altresì un caso di test che copre la possibilità di avere due triplette con valore di *Nome* uguale.

	Nome	Password	Username	CE coperte
<b>Test Case</b>				
TC1	"pippo"	"topolino"	"paperino"	ce1, ce5, ce9
TC2	"ç°§@#*+"	"topolino"	"paperino"	ce2, ce5, ce9
TC3	"123456"	"topolino"	"paperino"	ce3, ce5, ce9
TC4	""	"topolino"	"paperino"	ce4, ce5, ce9
TC5	"testv"	"ç°§@#*+"	"paperino"	ce2, ce6, ce9
TC6	"testvi"	"1234567"	"paperino"	ce2, ce7, ce9
TC7	"testvii"	""	"paperino"	ce2, ce8, ce9
TC8	"testviii"	"ç°§*+"	"ç°§*+"	ce2, ce6, ce10
TC9	"testix"	"ç°§*+"	"1234568"	ce2, ce6, ce11
TC10	"testx"	"ç°§*+"	""	ce2, ce6, ce12,
TC11	""	""	""	ce4, ce8, ce12
TC12	"pippo"	"1234567"	<u>"paperino"</u>	ce1, ce7, ce9

**Figura 7 Test Suite per il testing funzionale Black Box**

Sui campi di Output avremmo un comportamento speculare, nel senso che i valori validi e non validi sono gli stessi dei campi di input.

I test sono stati creati con l'ausilio del tool (Abbot&Costello) ed eseguiti da con il framework Junit su Eclipse dalla test suite CoperturaClassiTest.java

Tutti i casi di test della test suite costruita sonno falliti, questo è successo come conseguenza del fallimento delle asserzione inserite nei test case.

I test case TC4, TC7, TC10 e TC11 vanno in errore di TimeOut poiché essi prevedono uno o più campi vuoti e dopo il click sul buttonne di avviso in Figura 8 Avviso campi vuoti



Figura 8 Avviso campi vuoti

Non è prevista la visualizzazione dell'altro pulsante di conferma di avvenuta memorizzazione dei dati come presentato in Figura 9 Avviso dati salvati con successo



Figura 9 Avviso dati salvati con successo

Il test case **TC2** invece si pensa che sia andato in errore per un bug di Costello che interpreta in modo sbagliato i caratteri che sono stati usati per inserire il nome della tripletta. Viceversa, il **TC5** presenta una failure che non riesce a replicare in input i carattere "@" e "#".

Tutti gli altri test case hanno fallito per causa dell'asserzione inserita sul valore del Username e Password del elemento appena salvato e selezionato dalla lista. Si è notato che il valore dell'Username non era uguale all'Username inserito, bensì alla Password.

Con un'analisi del codice e con l'ausilio delle tecniche di debugging spiegate in seguito, è stato possibile localizzare e risolvere questi errori.

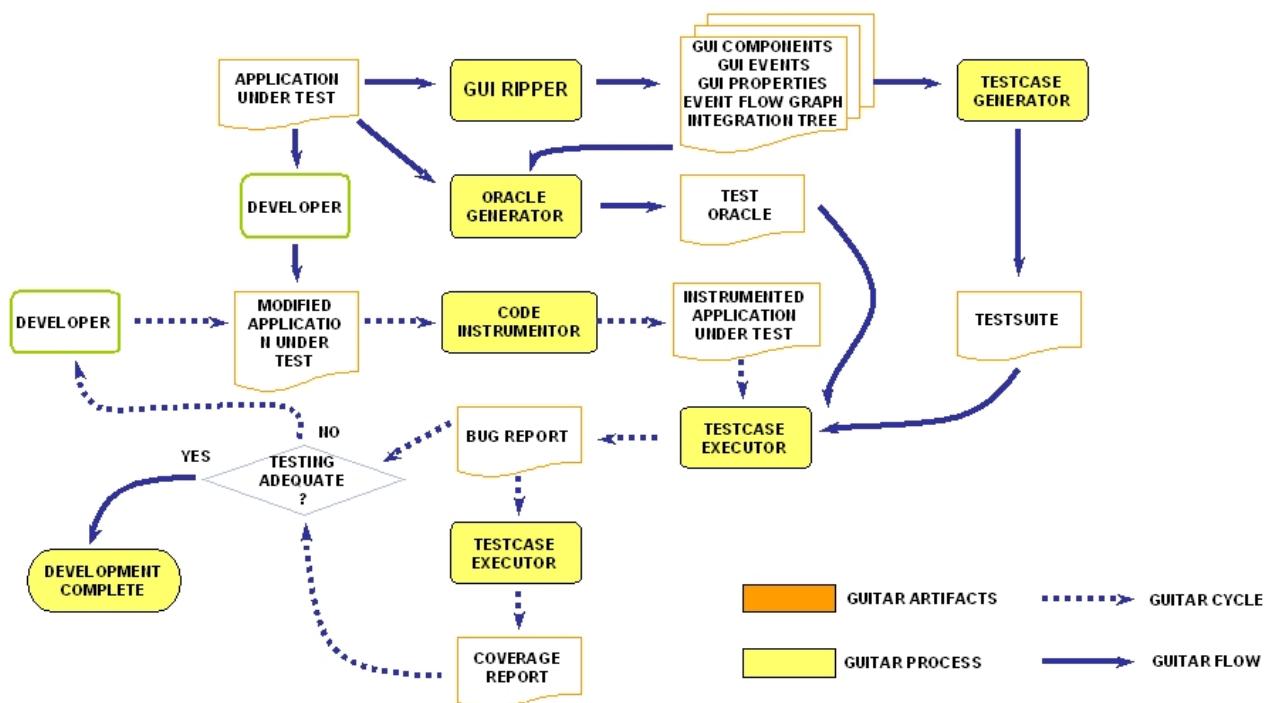
**TC3** presenta una failure per un nuovo malfunzionamento del programma. In pratica le password salvate sono ordinate per Nome ogni volta che si presenta un inserimento o una cancellazione di un elemento. In seguito, all'avvio del programma è preselezionato il primo degli elementi che erroneamente non è visualizzato e, nel caso in cui si desidera selezionare nuovamente non viene visualizzato lo stesso. Questo malfunzionamento porta il test case a fallire, poiché cerca di visualizzare il primo elemento della lista.

# Testing GUI don GUITARE

A scopo sperimentale è stata utilizzata una suite di tools (Guitar), basata sul concetto di GUI ripping.

**Il GUI RIPPING** è una tecnica di Reverse Engineering per l'automazione della generazione di casi di test e della loro successiva esecuzione.

La GUI è automaticamente attraversata in modo da identificare tutti i widget, finestre, campi di input, proprietà e valori che la costituiscono, in modo da ottenere una rappresentazione sotto forma di grafo delle dipendenze, a partire dalla quale vengono ricavati tutti i possibili scenari di interazione e relativi test cases. Questi ultimi sono automaticamente eseguiti e verificati.



**Figura 10 Funzionamento concettuale dei tool Guitare**

Con GUITARE dunque si risparmia il lavoro di scrivere test cases, ma questi vengono automaticamente generati e successivamente eseguiti in automatico sulla GUI.

Il Processo di testing con la suite tools GUITARE è suddiviso in 4 passi:

- 1. Ripping:** Vengono ricercati tutti gli elementi nascosti della GUI e viene ricavata la struttura della GUI attraverso il tool GUI Ripper. Viene dunque prodotto un **gui\_file**; un xml rappresentante la struttura della GUI.
- 2. Model construction:** Dalle informazioni ricavate precedentemente viene costruito un modello della GUI, rappresentato sottoforma di **Event-Flow Graph**.
- 3. Test case generation:** Vengono generati i Test Case attraverso l'esecuzione di un algoritmo di traversamento sul grafo generato in precedenza.
- 4. Reply:** I Test Cases sono eseguiti e verificati i risultati. Si generano due file:
  - a. **log\_file**: contiene tutte le informazioni relative all'esecuzione del test, eccezioni e messaggi.
  - b. **gui\_state**: che contiene tutte le informazioni inerenti agli stati degli elementi della GUI al momento dell'esecuzione del Test Case

Un **Event-Flow Graph** è un grafo orientato che modella tutte le possibili sequenze di eventi che possono essere eseguite sulla GUI. I nodi rappresentano gli eventi della GUI (es: click su un bottone) e gli archi rappresentano le relazioni tra gli eventi. Se un nodo D segue un dono S significa che l'evento rappresentato da D può essere eseguito immediatamente dopo l'evento S.

Per rieseguire il testing con GUITARE sul nostro programma, bisogna prima di tutto impostare ~~settare~~ facilmente il path dell'applicazione e gli altri pochi settaggi nei file:

- jfripper.properties
- GUIStructure2GraphConvert.properties
- TestCaseGenerator.properties
- jfcreplayer.properties

dopo di che eseguire in ordine:

- ant -Dproperties=jfripper.properties -f jfripper.xml
- ant -v -f GUIStructure2GraphConvert.xml
- ant -v -f TestCaseGenerator.xml
- ant -Dproperties=jfcreplayer.properties -f jfcreplayer.xml

Nel file TestCaseGenerator.properties si può settare il numero massimo di test cases che si vogliono generare, al momento è settato a 0 in modo da creare tutti i TC possibili.

È stato eseguito il tool in modo da creare tutti i test case possibili e sono stati creati in automatico 575 test cases. Conseguentemente si è deciso di dare come feed al player solo una parte di questi test case. Nel file Frame.log si può osservare l'esito dei test case eseguiti.

# Verifica della copertura White-box

## Copertura Testing della GUI

Dai Test Cases realizzati per il testing della GUI è stata verificata la copertura White-Box mediante lo strumento (CodeCover). Le statistiche fanno riferimento alla copertura dei comandi, delle condizioni e delle decisioni.

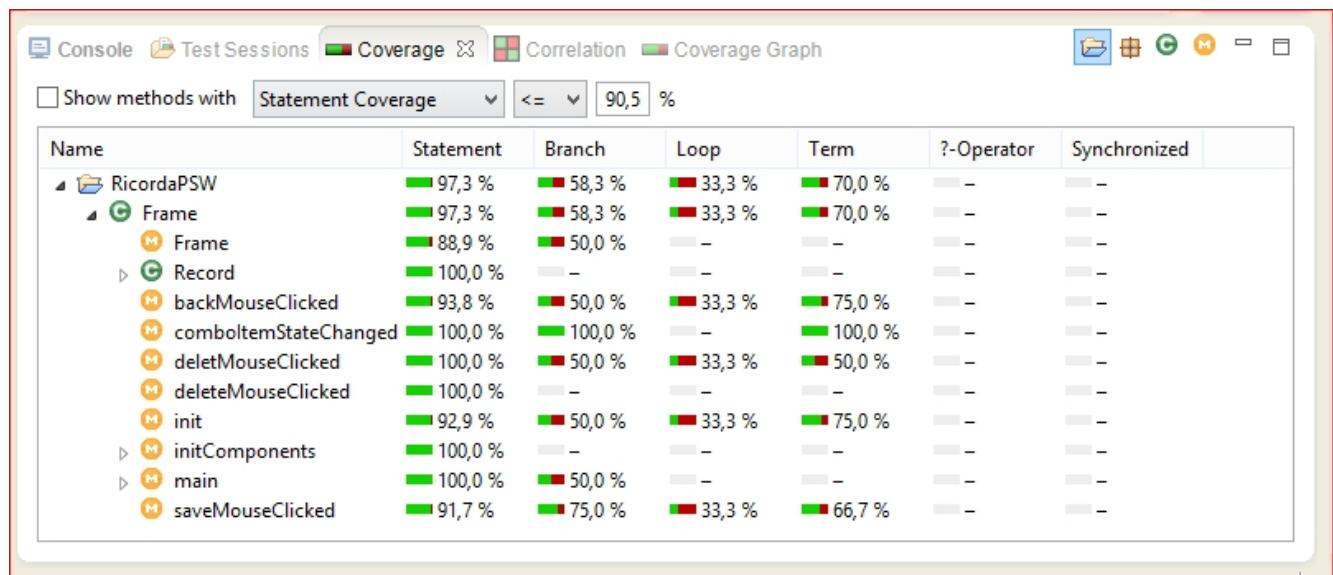


Figura 11 Copertura della classe Frame.java con il GUI testing

Come si evince dal report in Figura 11 Copertura della classe Frame.java con il GUI testing i test effettuati hanno consentito di avere una buona copertura dei metodi della classe Frame.java e una copertura meno ottimale per i altri criteri analizzati.

## Copertura testing Funzionale

Invece la copertura ottenuta dall'esecuzione dei test cases progettati per il **testing black box** viene riportato in Figura 12 Copertura della classe Frame.java attraverso il testing BlackBox. Si può notare una più bassa copertura generale, come si poteva aspettare, poiché questa test suite sollecita solo l'operazione di inserimento e di cancellazione di item e una migliore copertura dei cicli. Il metodo `backMouseClicked()` in questo caso non è stato coperto.

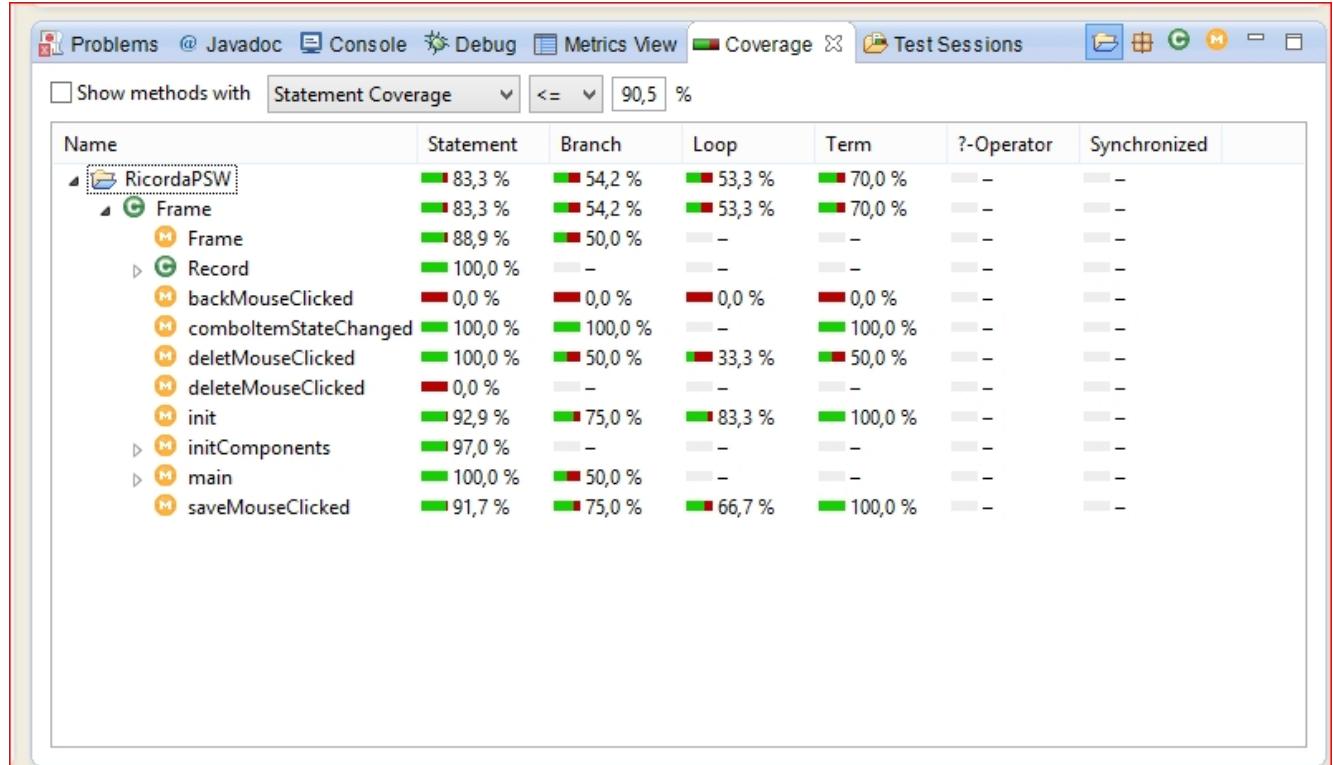


Figura 12 Copertura della classe Frame.java attraverso il testing BlackBox

La copertura non ottimale dei metodi viene anche dal fatto che CodeCover considera nel calcolo della copertura anche il codice di gestione delle eccezioni che, ovviamente, non è coperta dalla test suite e come si può notare nel metodo `saveMouseClicked()` qui riportato non viene coperto.

```

private void saveMouseClicked(java.awt.event.MouseEvent evt) { //GEN-FIRST:event_saveMouseClicked
    if ((home.getText().equals("")) || (user.getText().equals("")) || (psw.getText().equals("")))
        JOptionPane.showMessageDialog(null, "Devi riempire tutti i campi banana!", "Attenzione", JOptionPane.INFORMATION_MESSAGE);
    }
    else //mancante
}

```

```
{  
    try {  
        lista.add(new Record(nome.getText(), user.getText(),  
psw.getText()));  
        fout = new PrintWriter(new FileWriter(path));  
        for (Record x : lista) {  
            fout.println(x.toRecord());  
        }  
        fout.close();  
        init();  
        user.setText("");  
        nome.setText("");  
        psw.setText("");  
        nome.requestFocus();  
        JOptionPane.showMessageDialog(null, "Dati salvati con successo  
:", "Messaggio", JOptionPane.INFORMATION_MESSAGE);  
    } catch (Exception e) {  
        System.out.println(e.getMessage());  
    }  
}  
}//GEN-LAST:event_saveMouseClicked
```

# Debugging e Testing di Regressione

## Debugging

In seguito all'attività di esecuzione delle test suite, sono stati individuati **4 malfunzionamenti** descritti in seguito:

1. Il primo riguarda la possibilità di eliminare un elemento dalla lista delle password salvate. L'eliminazione non veniva effettuata anche se il messaggio di conferma veniva visualizzato. Il test case *RpswGUI2* rivela questo malfunzionamento attraverso l'asserzione inserita in esso.

Il **debugging** di questo malfunzionamento è stato effettuato utilizzando una strategia di *back tracking*. Dal punto in cui è visualizzato il messaggio conferma eliminazione dell'elemento all'interno del metodo `deleteMouseClicked` andando a ritroso è stato possibile individuare il difetto.

Eseguendo l'applicazione in modalità debug è stato localizzato il difetto. Questo era rappresentato da un'istruzione mancante, che si doveva occupare dell'eliminazione dell'elemento scelto e dalla Lista utilizzata come struttura dati all'interno del applicazione.

## CODICE ERRATO

```

if (scelta == 0) {
    try {
        fout = new PrintWriter(new FileWriter(path));
        for (Record x : lista) {
            fout.println(x.toRecord());
        }
        fout.close();
        init();
        JOptionPane.showMessageDialog(null, "Record eliminato !! :)",
        "Messaggio", JOptionPane.INFORMATION_MESSAGE);
    } catch (Exception e) {
    }
}

```

## CODICE CORRETTO

```

if (scelta == 0) {
    try {
        fout = new PrintWriter(new FileWriter(path));
        lista.remove(p); //istruzione mancante
        for (Record x : lista) {
            fout.println(x.toRecord());
        }
        fout.close();
        init();
        JOptionPane.showMessageDialog(null, "Record eliminato !! :)",
        "Messaggio", JOptionPane.INFORMATION_MESSAGE);
    } catch (Exception e) {
    }
}

```

## DESCRIZIONE:

Il difetto era dovuto alla mancanza dell'istruzione che si doveva occupare di eliminare dalla struttura dati *lista* l'elemento selezionato dal utente. La *lista* che conteneva tutte le triplette *nome;username; password* veniva ricopiata all'interno del file senza eliminare prima tale elemento. All'interno della condizione if si entra nel caso in cui l'utente esegue un click sul pulsante **Si** della finestra di dialogo (`JOptionPane.showConfirmDialog "Sei sicura di voler eliminare: "`) questa scelta veniva percepita in modo corretto dal programma, assegnando alla variabile *scelta* il valore corretto 0.

2. La seconda failure invece riguarda l'erroneo salvataggio di un elemento che presenta uno dei campi “Nome”, “Username” o “Password” vuoti.

Essendo l'applicazione non di grosse dimensioni e lavorando in ambiente Eclipse, si è scelto di lavorare in modalità “forza brutta”, inserendo all'interno del codice dei punti di break. In questo modo il programma si manda in esecuzione con la peculiarità che si arresta ogni volta si incontra un breakpoint.

Sono stati inserito diversi breakpoint nel codice, seguendo il metodo `saveMouseClicked` che è responsabile del controllo dei valori dei campi *Nome*, *Password* e *Username* e del loro salvataggio. Eseguendo l'applicazione in modalità debug è stato localizzato il difetto all'interno di questo metodo. Anche in questo caso si è trattato di un'istruzione mancante.

### CODICE ERRATO

```

private void saveMouseClicked(java.awt.event.MouseEvent evt) { //GEN-FIRST:event_saveMouseClicked
    if (nome.getText().equals("") || user.getText().equals("") || psw.getText().equals(""))
        JOptionPane.showMessageDialog(null, "Devi riempire tutti i campi banana:", "Attenzione", JOptionPane.INFORMATION_MESSAGE);
    }{

    try {
        lista.add(new Record(nome.getText(), user.getText(), psw.getText()));
        fout = new PrintWriter(new FileWriter(path));
        for (Record x : lista) {
            fout.println(x.toRecord());
        }
        fout.close();
        init();
        user.setText("");
        nome.setText("");
        psw.setText("");
        nome.requestFocus();
        JOptionPane.showMessageDialog(null, "Dati salvati con successo :)", "Messaggio", JOptionPane.INFORMATION_MESSAGE);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}

```

**CODICE CORRETTO**

```

private void saveMouseClicked(java.awt.event.MouseEvent evt) {//GEN-FIRST:event_saveMouseClicked
    if (nome.getText().equals("") || user.getText().equals("") || psw.getText().equals("")) {
        JOptionPane.showMessageDialog(null, "Devi riempire tutti i campi banana:)
        !!", "Attenzione", JOptionPane.INFORMATION_MESSAGE);
    }
    else //mancante
    {

        try {
            lista.add(new Record(nome.getText(), user.getText(), psw.getText()));
            fout = new PrintWriter(new FileWriter(path));
            for (Record x : lista) {
                fout.println(x.toRecord());
            }
            fout.close();
            init();
            user.setText("");
            nome.setText("");
            psw.setText("");
            nome.requestFocus();
            JOptionPane.showMessageDialog(null, "Dati salvati con successo :)",
            "Messaggio", JOptionPane.INFORMATION_MESSAGE);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}

```

**DESCRIZIONE:**

Il difetto era dovuto al non completamento della struttura di controllo condizionale if-else. Infatti l'espressione all'interno del if controlla se i tre i campi di input sono diversi da stringa vuota. In caso affermativo essa stampa il messaggio di errore e il flusso di esecuzione dovrebbe terminare. Invece, in mancanza del else l'esecuzione, va avanti ugualmente, salvando i dati parziali. Questo errore era il motivo del malfunzionamento che comportava il passaggio per lo stato *Alert Conferma Operazione* come si può notare in Figura 2 Parte della FSM usate per progettare i test cases RpswGUI1, RpswGUI2 e RpswGUI3.

La correzione di questo secondo malfunzionamento ha comportato la mutazione del comportamento della GUI, poiché la copertura dei path evidenziati in rosso in Figura 3 Parte della FSM usate per progettare i test cases RpswGUI4 e RpswGUI5Figura 2 Parte della FSM usate per progettare i test cases RpswGUI1, RpswGUI2 e RpswGUI3 che prevedeva il test case RpswGUI4 non è più valida e non viene più visualizzato lo stato “*Alert Conferma Operazione*” come prima.

Per rieseguire i test è stato necessario modificare leggermente i su detti test casi per togliere il click sul pulsante di conferma “*Dati salvati con successo :)*”

3. Il prossimo malfunzionamento che è stato messo in mostra dai Test Case TC1, TC3, TC5 TC6, TC8 TC9 riguardante un'inversione dei campi di input “Username” e “Password”. Per risolverlo ci siamo avvalse della funzionalità di Costello che mostra i nomi delle instance JText Fields che compongono la GUI.

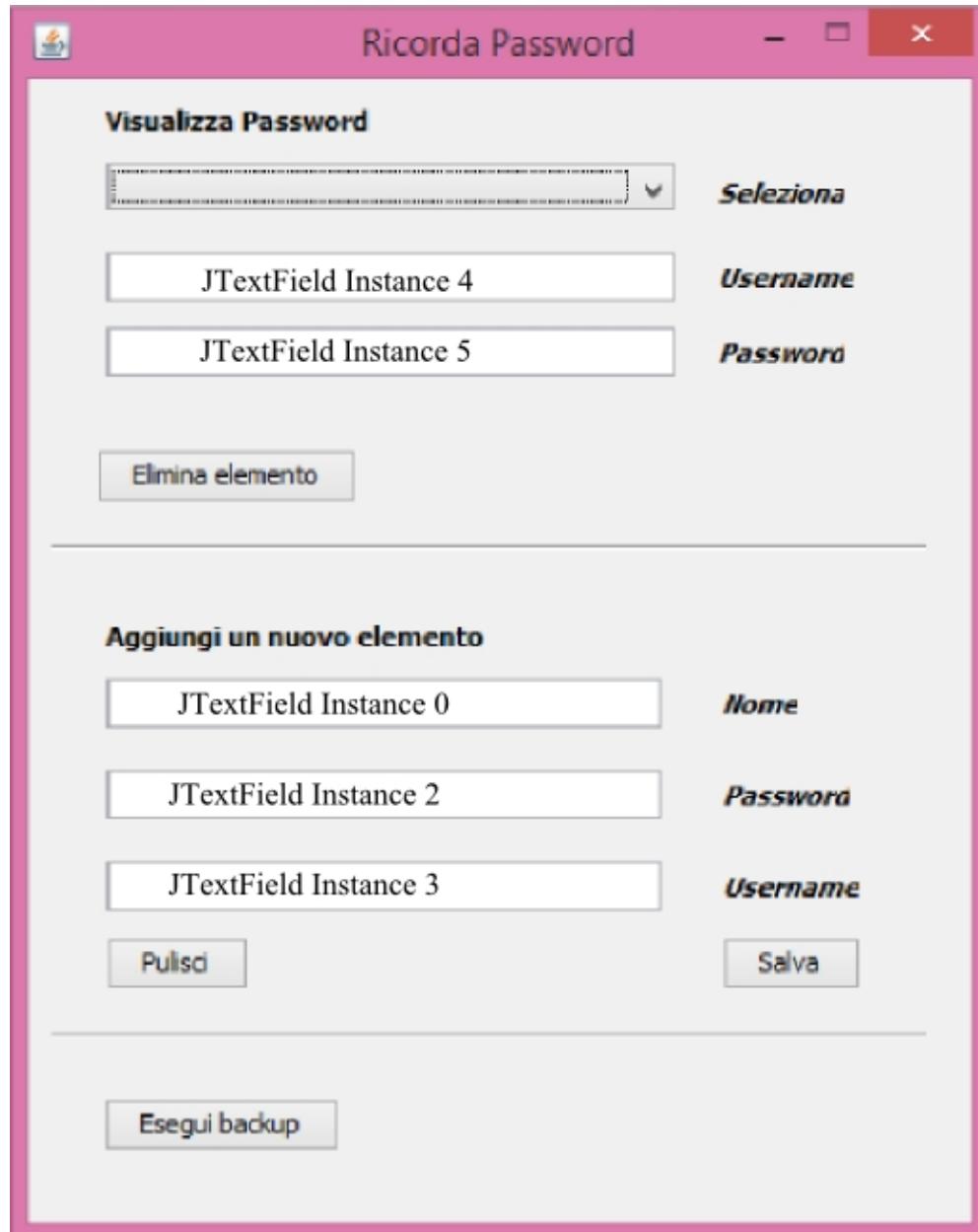


Figura 13 Ispezione della GUI con Costello

Questo non è servito poiché si è notato che non c'è una corrispondenza univoca tra le istanze visualizzate dal tool e gli elementi definiti nel programma.

La correzione della failure è stata resa possibile solo attraverso l'analisi statica del codice. Difatti utilizzando la tecnica di Code Reading (detta anche Desk Checking) è stato possibile individuare la porzione di codice che ha causato tale anomalia.

### CODICE ERRATO

```
user.setFont(new java.awt.Font("Tahoma", 0, 12)); // NOI18N
jLabel6.setFont(new java.awt.Font("Tahoma", 3, 11)); // NOI18N
jLabel6.setText("Username");

psw.setFont(new java.awt.Font("Tahoma", 0, 12)); // NOI18N

jLabel5.setFont(new java.awt.Font("Tahoma", 3, 11)); // NOI18N
jLabel5.setText("Password");

nome.setFont(new java.awt.Font("Tahoma", 0, 12)); // NOI18N
```

### CODICE CORRETTO

```
user.setFont(new java.awt.Font("Tahoma", 0, 12)); // NOI18N
jLabel5.setFont(new java.awt.Font("Tahoma", 3, 11)); // NOI18N
jLabel5.setText("Username");

psw.setFont(new java.awt.Font("Tahoma", 0, 12)); // NOI18N

jLabel6.setFont(new java.awt.Font("Tahoma", 3, 11)); // NOI18N
jLabel6.setText("Password");

nome.setFont(new java.awt.Font("Tahoma", 0, 12)); // NOI18N
```

### DESCRIZIONE:

Il difetto era dovuto all'uso erroneo dei due elementi JLabel della GUI, jLabel5 e jLabel6 i quali vengono visualizzati in concomitanza dei componenti JTextField sbagliati. È stato necessario l'inversione della jLabel5 con la jLabel6 per correggere l'errore ed eliminare il malfunzionamento.

4. Infine è stato scoperto un ulteriore malfunzionamento grazie al **TC3**. In pratica il programma non permette di selezionare la prima tripletta dalla lista senza aver visualizzato prima un'altra. Nel caso in cui abbiamo solo una password salvata, non è possibile visualizzarla.

Analizzando il codice è stato possibile individuare il metodo responsabile della visualizzazione dell'Username e della Password ma non si sono riscontrati anomalie. Bisogna fare un'analisi più dettagliata della struttura del programma per risolvere questo errore.

## Testing di regressione

In seguito alla correzione degli errori sono stati eseguiti i test suite per Costello, presenti nei package “*gui\_suite\_testing*” e “*blackbox\_suite\_testing*”

Come si può vedere nella Figura 14 risultati della esecuzione di CostelloTest suite gli errori individuati dalla test suite CostelloTest sono stati risolti e non ci sono state introduzioni di nuovi malfunzionamenti.

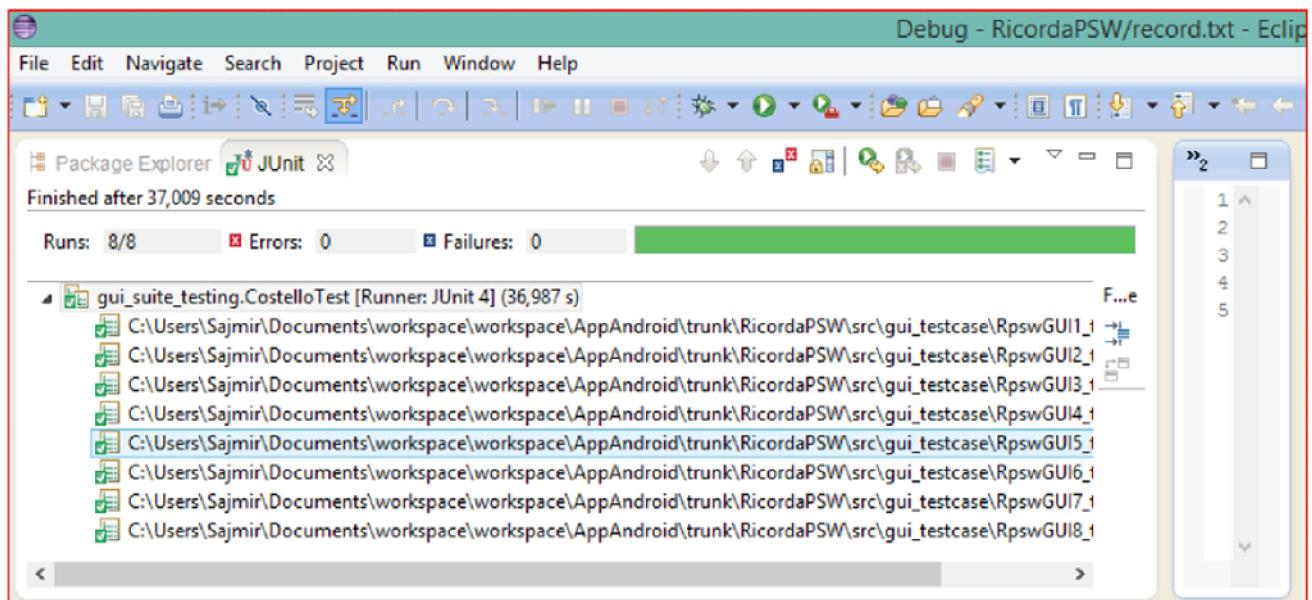


Figura 14 risultati della esecuzione di CostelloTest suite

Invece, come si può evincere dalla Figura 15 risultati della esecuzione di CoperturaClassiTest suite ora la test suite in questione presenta un errore in corrispondenza del test case TC2, poiché Costello non riesce ad replicare l'inserimento in input dei caratteri @ e # e per questo motivo non trova tra le password salvate, quella che precedentemente è stato salvato. Invece il TC5 per lo stesso malfunzionamento presenta una failure poiché l'asserzione non diventa vera, non avendo inserito i caratteri suddetti.

**TC3** viceversa riguarda il malfunzionamento presentato nel 26 precedente come quarto errore scoperto.

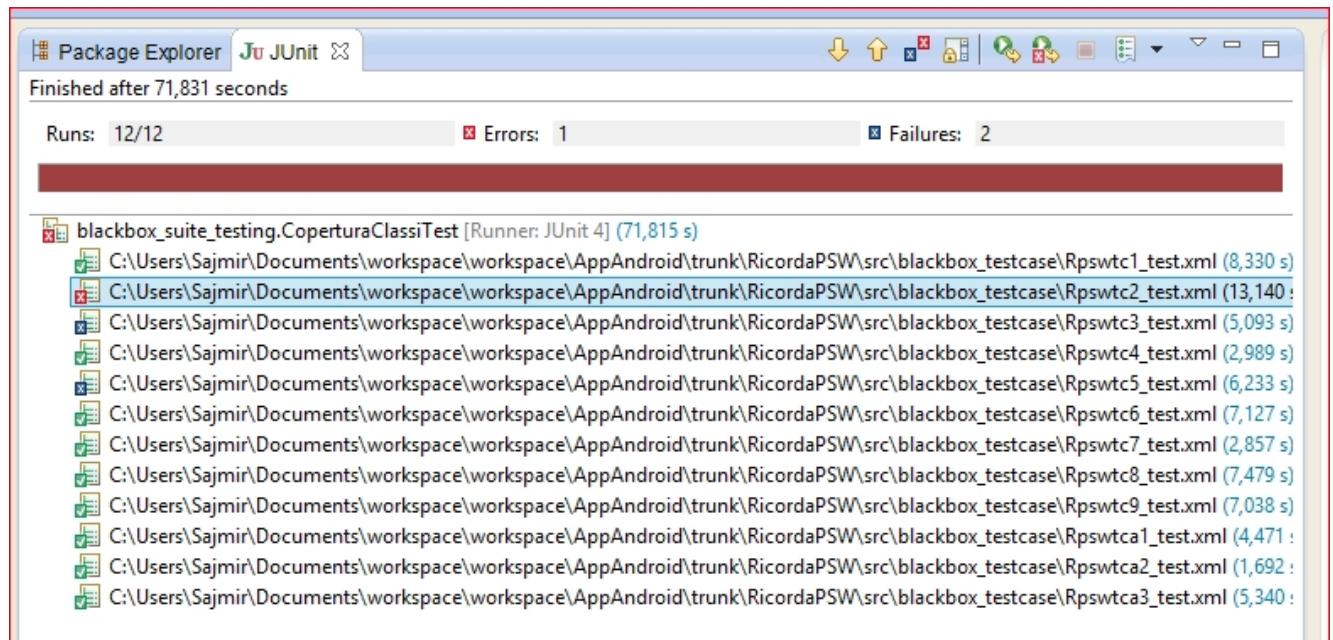


Figura 15 risultati della esecuzione di CoperturaClassiTest suite

## Bibliografia

- Costello, A. (s.d.). *Abbot framework for automated testing of Java GUI components and programs.* Tratto da <http://abbot.sourceforge.net/doc/overview.shtml>
- Guitar. (s.d.). *GUI Testing FrAmewoRk (GUITAR).* Tratto da <http://www.cs.umd.edu/~atif/GUITAR-Web/index.html.old>
- Tobias : Fourier, U. J. (s.d.). *TOBIAS.* Tratto da TOBIAS : a tool for combinatorial testing: <http://tobias.liglab.fr/>

Black Box Testing	
Debugging.....	22
Debugging e Testing di Regressione .....	22
Introduzione .....	3
<b><i>User Interface Testing</i></b> .....	5
Testing Funzionale.....	13

Testing Brack Box delle Partizioni .....	13
<b>Testing GUI don GUITARE</b> .....	16
Verifica della copertura White-box.....	18
Copertura Testing della GUI .....	18
Copertura testing Funaionzle .....	19