Software Testing

User Interface Testing

Interfacce utente

- **Le interfacce utente possono essere classificate in tre tipologie fondamentali:**
  - **Interfacce utente a carattere (CUI): sono le classiche interfacce utilizzate dalle shell dei sistemi operativi. In esse gli input arrivano tramite uno stream di input**
  - **Interfacce utente form-based: sono utilizzate in alcuni classici calcolatori IBM e in molti programmi vecchi (ad esempio il BIOS); sono ad esse analoghe anche le interfacce delle pagine web, almeno se escludiamo le interazioni con mouse o altri dispositivi di puntamento. Nelle interfacce form-based, gli input arrivano tramite uno stream nel quale ci sono sia caratteri di input che caratteri speciali (tabulazioni, backspace, tasti funzione, etc.)**

Interfacce GUI

- **Le interfacce utente possono essere classificate in tre tipologie fondamentali:**
  - **Interfacce utente grafiche (GUI): sono le interfacce più utilizzate nei PC. In esse gli input arrivano tramite uno stream di eventi, comprendenti eventi da tastiera (tasti premuti), eventi da altri dispositivi di puntamento (mouse, touch pad, touch screen, etc.). Tutti questi eventi confluiscono in uno stream, nel quale vengono interpretati e vengono riconosciuti eventi di più alto livello.**
    - **Ad esempio un doppio click si ottiene da uno stream di eventi nel quale si notano due coppie di operazioni di premuta e rilascio del pulsante sinistro del mouse, avvenute a distanza ravvicinata di tempo e su pixel ravvicinati sullo schermo**

Sistemi basati sugli eventi

- **Le interfacce utente grafiche rappresentano un caso di sistema ad eventi**
  - Il sistema è in uno stato stabile fino all'intervenire di un evento utente, che fa partire un codice di event handling
- **Anche un calcolatore, dotato di sistema delle interruzioni, può essere considerato come un sistema ad eventi**
  - Un segnale è in grado di avviare un'interruzione, che viene successivamente servita
- **I moderni sistemi distribuiti a sensori devono essere considerati sistemi ad eventi**
  - Ogni dispositivo è in grado sia di ricevere input, che di elaborarli

Testing di sistemi interattivi

- **Il testing di sistemi interattivi (in particolare di interfacce utente) viene condotto tipicamente in maniera black box**

- **Il testing di sistemi interattivi è utilizzato:**
  - **In caso di testing in isolamento del codice relativo all'interfaccia utente**
  - **In caso di testing di integrazione (con strategia top-down, ad esempio)**
  - **In caso di testing di sistema: l'interfaccia utente è il punto d'accesso al sistema**

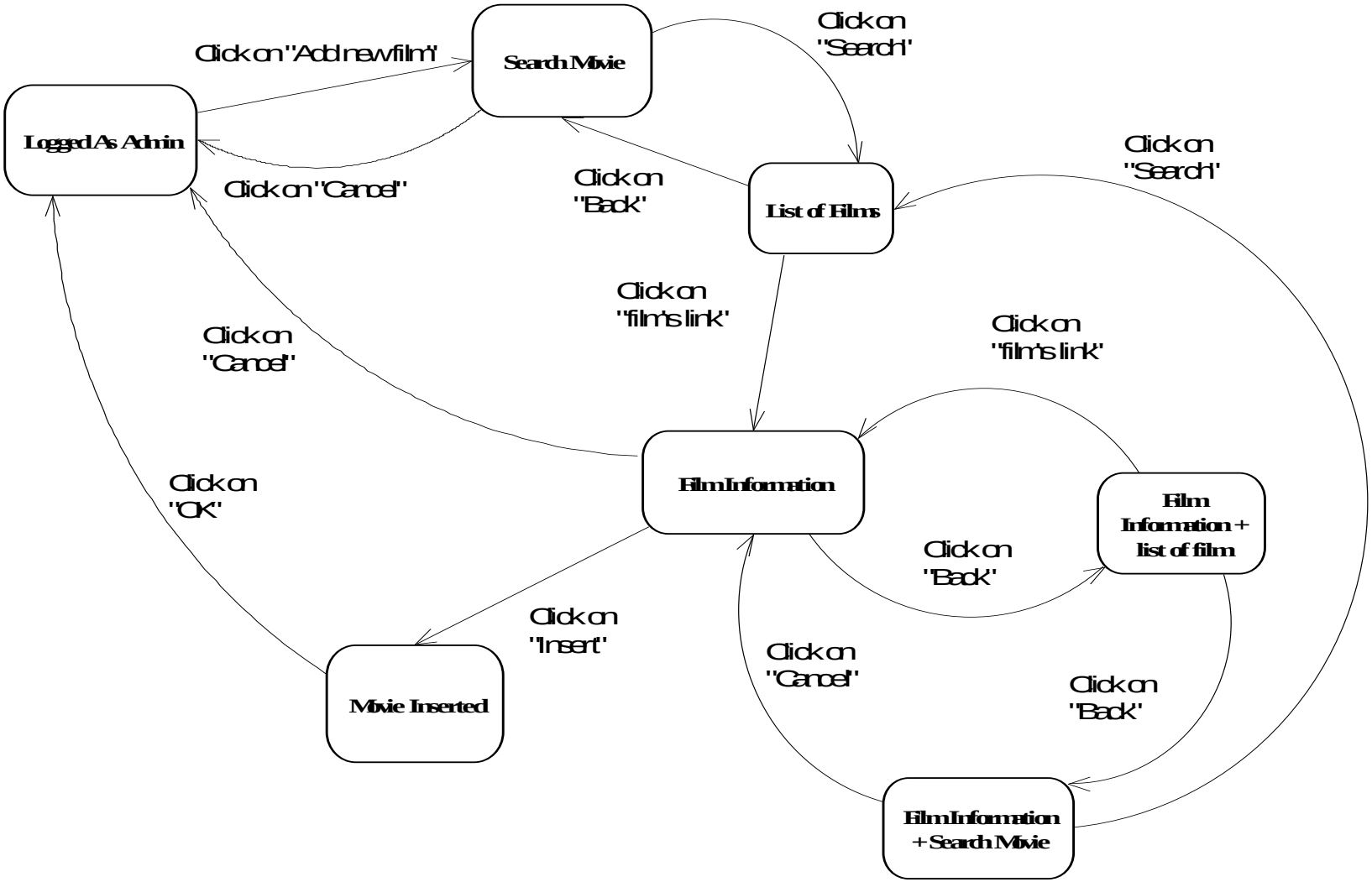
Testing di sistemi interattivi

- **Tipicamente i casi di test sono progettati tenendo in conto le possibili interazioni che un utente può eseguire sull'interfaccia utente**
  - Per poter approcciare il problema del testing è fondamentale la disponibilità di un modello descrittivo delle interazioni utente-macchina
  - Quali modelli sono utilizzabili per modellare UI?

Modellazione delle interazioni

- **Il modello più comune è un modello di Macchina a stati:**
  - Gli elementi fondamentali di una UI sono:
    - ***Finestre e relativi Widget, dotati di***
      - **Eventi che possono essere eseguiti dagli utenti**
      - **Campi, che possono essere eventualmente settati**
- **Si suppone che l'interfaccia utente non esegua alcuna operazione se non in seguito a sollecitazioni da parte degli utenti**
  - Lo "**stato**" dell'interfaccia utente viene modellato da uno stato di un automa
  - L'esecuzione di **eventi** sulla UI (es. Click su button…) viene modellata come ingressi impulsivi che scatenano transizioni nell'automa
  - Gli **input** immessi sono modellati come ingressi a livelli da cui dipende la transizione che si verifica

Esempio- modello di FSM per una UI



Es.: L'FSM che descrive l'evoluzione di una UI per l' esecuzione di un Caso d'Uso

Testing delle interfacce

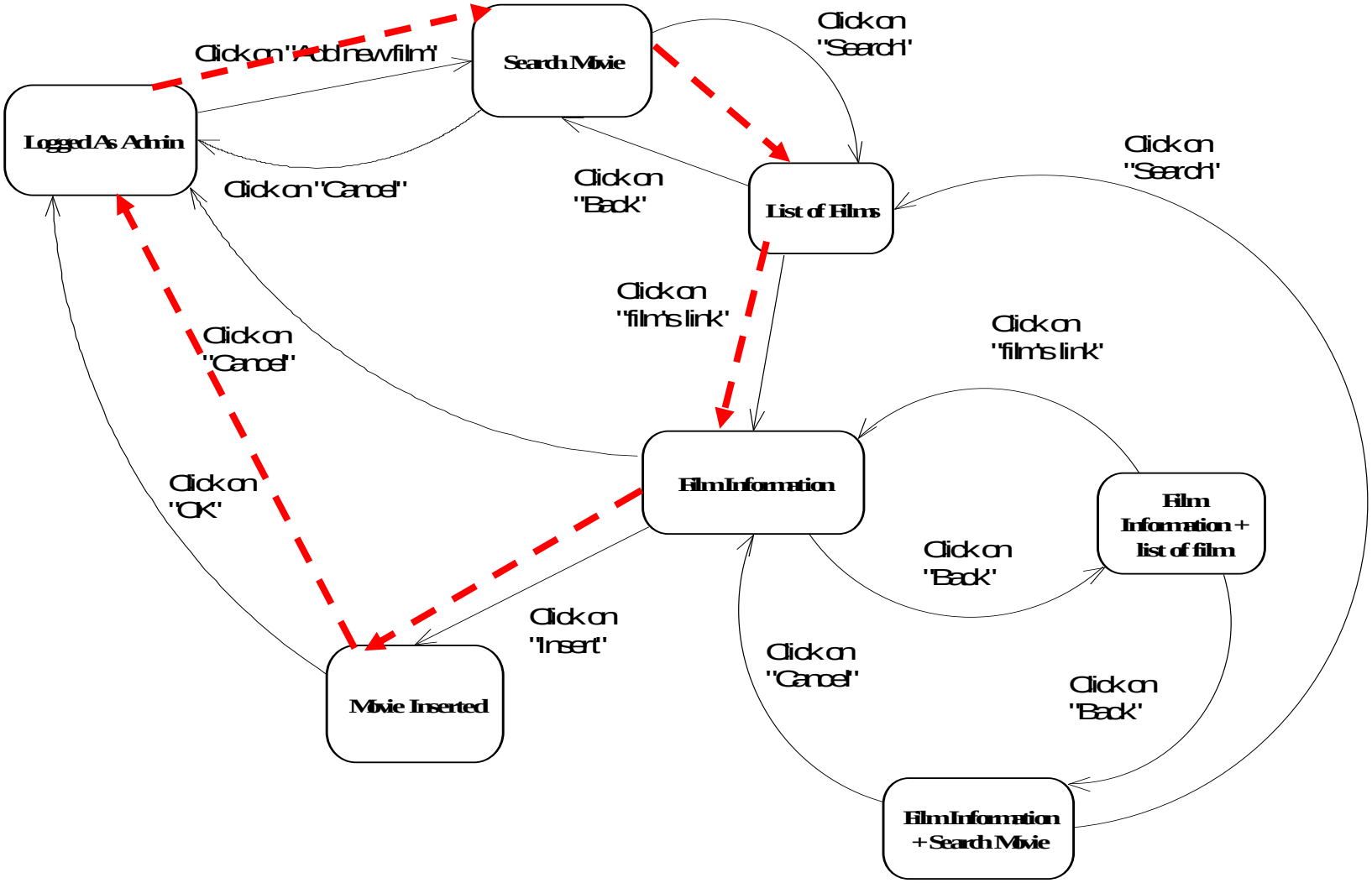
- Rispetto al modello riportato come esempio vanno aggiunti:
  - Lo stato iniziale, da cui si raggiunge la prima schermata dell'applicazione
  - Lo stato finale, raggiungibile di solito da qualsiasi altro stato (premendo il pulsante di chiusura dell'applicazione)

Testing delle interfacce

- Gli input dei casi di test devono essere indicati sotto forma di **sequenze** di **valori di input** ed **eventi** da eseguire.
  - A seconda dei criteri di copertura (es. Stati, Eventi, Transizioni, path…), si progetteranno diversi TC
  - **Es.: click (add-film),click(search), click(film link),click(insert), click(OK)** è il TC che permette di eseguire il cammino in rosso (*Scenario: Inserimento Film OK*)

- Ad una sequenza di valori di input dovrebbe corrispondere una **sequenza** di **stati visitati** e di **output** riscontrati.

Esempio- modello di FSM per una UI



Es.: Cammino di esecuzione che esercita un certo scenario del Caso d'Uso

Analogia

- In analogia con il testing white box, nel quale cercavamo di effettuare la copertura del grafo ciclico CFG, qui abbiamo di base gli stessi obiettivi e problemi:
  - Coprire tutti i nodi della FSM
  - Coprire tutti gli archi della FSM
  - Coprire tutti i cammini linearmente indipendenti della FSM
  - Coprire tutti i cammini della FSM
    - Al raggiungimento automatico di ognuna di queste coperture corrisponde un problema indecibile

Ulteriore problema

- Come ottenere un FSM che descriva adeguatamente l'UI e che possa essere usato per progettare i casi di Test?
- Due possibilità:
  - FSM prodotto in fase di sviluppo dell'applicazione (ma può essere poco aderente all'effettiva implementazione fatta dell'UI)
  - FSM ricostruito per Reverse Engineering a partire dalla UI effettivamente implementata (richiede tecniche di analisi statica o dinamica)
    - Nel caso di Interfacce Dinamicamente Configurabili (es. di Rich Internet Applications) piuttosto che Statiche, l'analisi statica non è sufficiente! Necessità di definire tecniche e strumenti di RE per la generazione (semi-automatica) del modello.

Esempio in Java

```
public JMenu createOptionMenu()
{
    JMenu m = new JMenu("Scientifica");
    JMenuItem item = new JMenuItem();

    class itemListener implements ActionListener
    {
         public void actionPerformed(ActionEvent e)
         {
              try{
                   op1 = Double.parseDouble(text.getText());
              }
              catch(NumberFormatException ecc)
              {
                   text.setText("Errore: nessun valore inserito!");
                   return;
              }
              operazione = RADICE_QUADRATA;
              invio.doClick();
         }
    }
…
}
```

Menu generato a run time

Voce di Menu generata a run time

Ascoltatore dichiarato a run time

Codice dell'ascoltatore dichiarato a run time

Non è, in generale, possibile, scrivere metodi di test che si riferiscono a oggetti, classi e metodi che vengono dichiarati a run-time

Altra analogia e altro problema

- La FSM ricavata per analisi statica del codice potrebbe non avere tutti gli elementi della GUI che poi vengono istanziati a tempo di esecuzione
  - E' lo stesso problema che si era riscontrato nel testing di integrazione per il Call Dependency Graph (grafo delle chiamate), a causa della possibile presenza di chiamate polimorfe, puntatori a funzione oppure collegamenti con librerie dinamiche

- Inoltre, nel FSM non dovrebbe essere rappresentata un'interfaccia staticamente definita (ad esempio una schermata con dei pulsanti), ma anche il suo stato (ad esempio anche i valori scelti all'interno di un form e i valori dei testi mostrati a video)
  - In questo modo il numero di nodi della FSM è praticamente sempre illimitato
    - Bisogna proporre delle tecniche per limitarlo

Problema degli stati equivalenti

- **Per poter astrarre un modello a stati è necessario poter decidere quando due stati coincidono**
  - Siccome qualsiasi valore di qualsiasi variabile può caratterizzare lo stato, il numero di stati possibili è potenzialmente illimitato
  - Per poter implementare dei criteri di equivalenza è necessario selezionare piccoli sottoinsiemi di informazioni, ma così facendo si rischia di non considerare informazioni importanti
    - *Nel nostro caso, una tecnica consiste nel considerare solo informazioni relative all'interfaccia utente, non allo stato interno dell'applicazione*
- **Da un modello FSM è possibile ricavare test suite che vanno a coprire tutti i nodi o tutti gli archi**
  - Un oracolo è dato dall'equivalenza dello stato raggiunto in ogni istante del test con quello descritto dal modello

Testing White Box di GUI

- **Il testing white box (o comunque la progettazione di classi di test di unità) per GUI è reso difficile da alcuni fattori:**
  - Molto spesso le classi della GUI sono istanziate a run-time, così come i metodi ascoltatori degli eventi
    - *Molto difficile scrivere dei driver che riescano ad emulare le stesse esecuzioni del programma originale*
  - Problemi di concorrenza: i test agiscono in un thread separato da quello che crea l'interfaccia, per cui bisogna anche misurarsi con le difficoltà relative alla tempificazione delle azioni di test

Java Robot (awt)

- La classe Robot consente l'esecuzione programmatica di eventi sull'interfaccia utente (limitatamente ad awt)
  - Ha metodi che riproducono le azioni del mouse ed eventi che riproducono la pressione di tasti
    - *Documentazione:* http://download.oracle.com/javase/1.4.2/docs/api/java/awt/Robot.html
    - *Un tutorial:* http://forum.codecall.net/java-tutorials/25923-robot-class.html

Esempio Java Robot

```
import java.awt.AWTException;
import java.awt.Robot;
import java.awt.event.InputEvent;

public class MovingMouseDemo {
   public static void main(String[] args) {
   try {
      Robot robot = new Robot();
      robot.mouseMove(200, 200);
      robot.mousePress(InputEvent.BUTTON1_MASK);
      robot.mouseRelease(InputEvent.BUTTON1_MASK);
      robot.mouseWheel(-100);
   } catch (AWTException e) {
   e.printStackTrace();
}
}
}
```

Il problema principale è quello di poter nominare gli oggetti dell'interfaccia (ad esempio InputEvent.BUTTON1_MASK) quando essi sono dinamicamente generati

"Twenty percent of all input forms filled out by people contain bad data."

- *Dennis Ritchie, More Programming Pearls: Confessions of a Coder by Jon Louis Bentley*

Validazione dell'input

- **Un tipico problema del testing delle interfacce utente è la verifica della validità dei dati di input**
  - E' la causa di molti attacchi (exploit) contro le applicazioni
  - Ad esempio, in linguaggi a puntatori, come C, l'immissione di una stringa troppo lunga in input, in mancanza di validazione, può portare a sovrascrivere altri dati o addirittura zone di codice del programma stesso
  - In linguaggi interpretati, come quelli spesso usati per il web, il problema è ancora più sentito

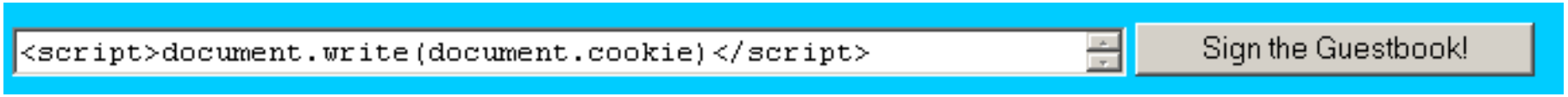
Il problema della validazione degli input

- La validazione dei dati può essere fatta sia sul lato client che sul lato server
  - La validazione sul lato client ha il vantaggio di utilizzare tempo di CPU della macchina client piuttosto che quello della macchina server
  - La validazione sul lato client può essere però scavalcata da un utente malintenzionato che vada a sollecitare il server con una richiesta http che non sia passata per la pagina client: in questo caso il server potrebbe avere delle anomalie

La soluzione più corretta è quella di porre la validazione sia sul lato client che sul lato server, in modo da bloccare la maggior parte delle richieste scorrette sul client (risparmiando risorse sul server). Solo le richieste fraudolente verrebbero così bloccate dalla validazione lato server

Esempio: Cross-Site Scripting

- Si verifica quando uno script lato client, inserito maliziosamente in un campo di input, viene eseguito sulla macchina client di un utente ignaro



password=MyPassword; login=Administrator; CurrentVersion=IT; LastVisit=20030502+10%3A36%3A17; ASPSESSIONIDAQTDQARB=HHKJLJJAONDNFOFJPLAAADDL

Sign.html

```
<form method="post" action="sign.asp">
    <textarea name="txtMessage"></textarea>
    <input type="submit" value="Sign!">
</form>
```

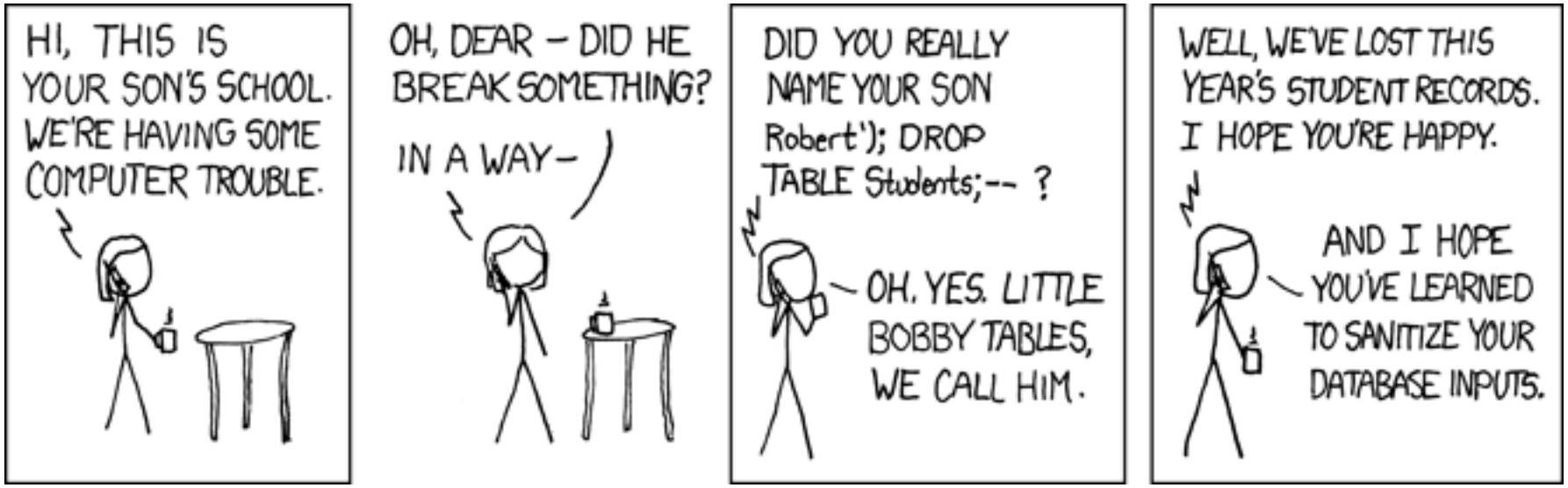
Message=Server.HtmlEncode(Message)

Sign.asp

```
<% Message=request.form("txtMessage")
   conn=OpenDBConnection
   set rs=server.createobject("Adodb.recordset")
   rs.open "Guestbook",conn,1,2,2
   rs.Addnew
   rs("Message")=Message
   rs.update
%>
```

Guestbook.asp

```
<% conn=OpenDBConnection
   rs=server.createobject("Adodb.recordset")
   rs.open "SELECT Message FROM GuestBook" ,
     conn,3,3
%>
<table>
<% rs.movefirst
   while not(rs.eof)
     response.write (rs.fields("Message"))
     rs.movenext
   wend
%>
</table>
<% rs.close
   set rs=nothing
   conn.close
   set conn=nothing
%>
```


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Testing basato su Sessioni Utente

- Tecnica per la generazione automatica di casi di test per il testing black box partendo dall'analisi delle sessioni utente (*User Session*), ovvero delle sequenze dei valori di input immessi e di output ottenuti in utilizzi reali del software.
  - In pratica, vengono installati strumenti che siano in grado di mantenere un ***log*** di tutte le interazioni che avvengono tra gli utenti dell'applicazione da testare e l'applicazione stessa (fase di ***Capture***)
  - A partire da tali dati vengono formalizzati casi di test che replichino le interazioni "catturate" (fase di **Replay**)
    - *In questo modo è possibile ottenere casi di test che siano rappresentativi dei reali utilizzi dell'applicazione da parte dei suoi utenti*
  - Il risultato ottenuto (in termini di output e stato) durante la fase di Capture può essere l'oracolo per i futuri Replay

Capture & Replay

- **Il sistema di Capture consiste in un ambiente di esecuzione controllato nel quale il tester «usa» l'applicazione e nel frattempo le sue interazioni sono registrate e «tradotte» in un caso di test rieseguibile**
- **E' possibile anche inserire delle asserzioni, in maniera visuale**

Limiti e problemi

- **Il sistema di Capture registra le interazioni, ma non lo stato del sistema: le precondizioni devono essere aggiunte dal tester**
- **Se il sistema ha delle race conditions, allora i casi di test generati devono tener conto anche della tempificazione degli input, altrimenti i test possono anche essere rieseguiti molto più velocemente rispetto alla velocità di registrazione**
- **Se cambia l'interfaccia utente, molti casi di test potrebbero non essere più eseguibili**

Window Tester Pro

- **Strumento di Capture e Replay per applicazioni Java con GUI realizzata con Swing**
  - Osserva un utente che esegue il programma e lo utilizza e registra il suo comportamento
  - Genera automaticamente codice Junit rieseguibile replicante le interazioni dell'utente

    I test ottenuti possono essere riutilizzati anche senza il supporto di Window Tester Pro
  - Aiuta l'utente nella scrittura di asserzioni

    Fornendo i nomi degli oggetti grafici a video

    Suggerendo alcuni possibili asserzioni

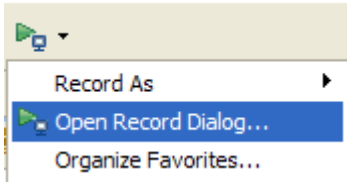
Window Tester Pro

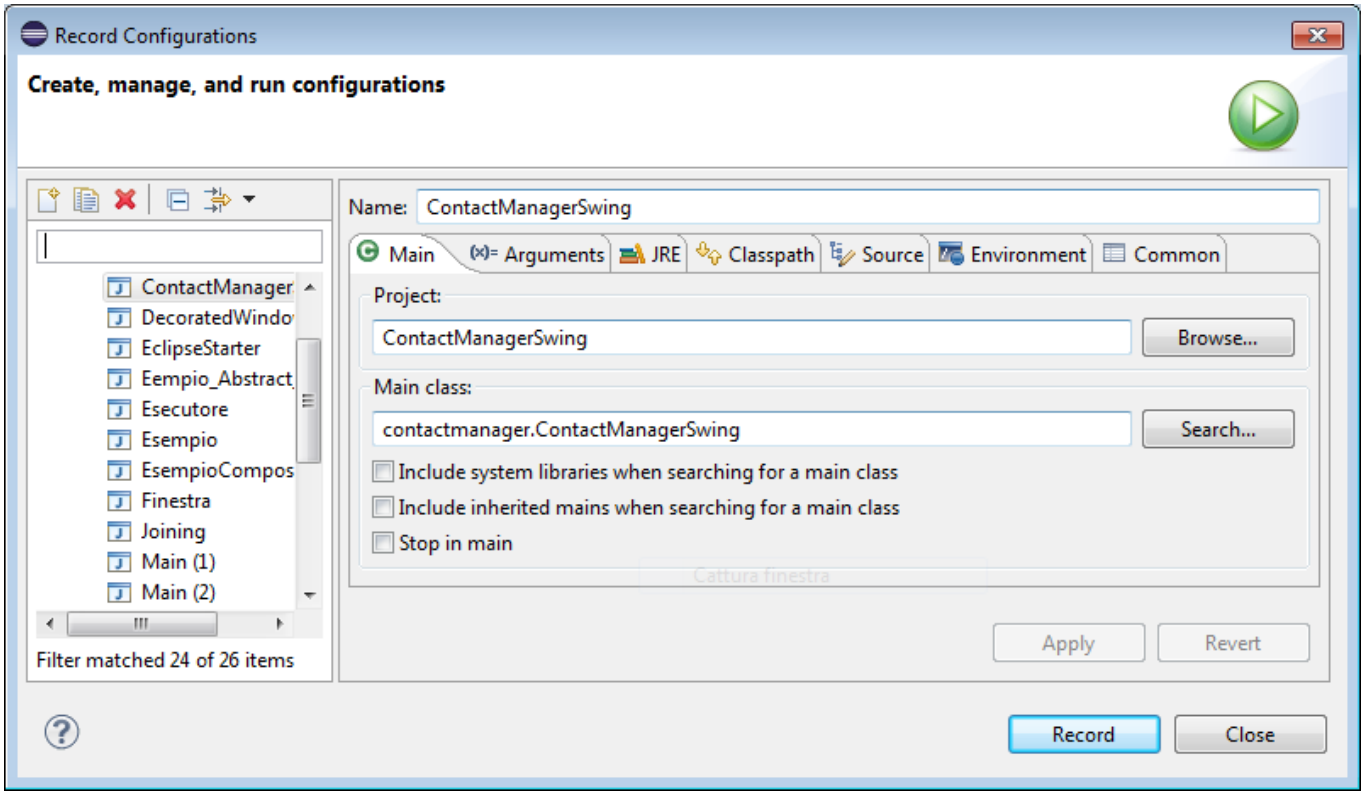
- **Installabile come plug-in di Eclipse**
  - http://dl.google.com/eclipse/inst/windowtester/latest/3.6
  - http://www.gwtproject.org/versions.html
- **Per eseguirlo bisogna:**
  - Creare un profilo di configurazione nell'ambito di Window Tester
  - Eseguire
  - Al termine dell'esecuzione vengono generati i test Junit

  Un tutorial completo è disponibile all'indirizzo:

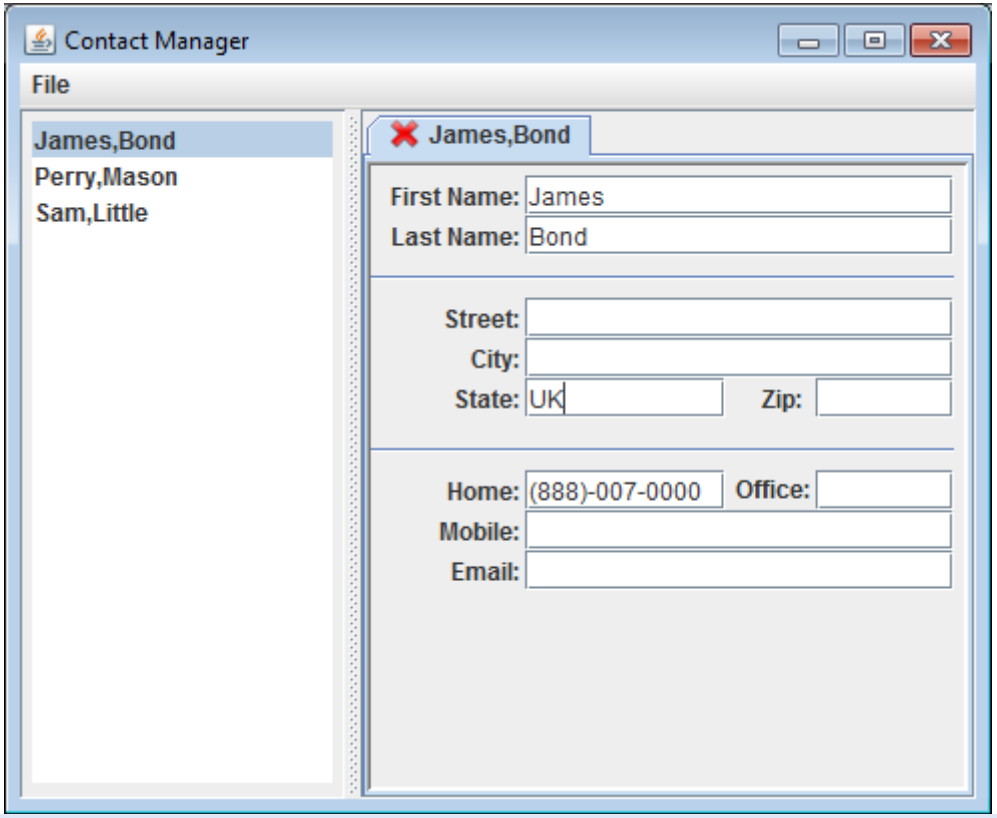
  - **https://developers.google.com/java-dev-tools/wintester/html/gettingstarted/swing_sampletest**

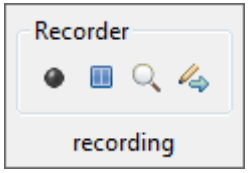
Creazione Profilo di Esecuzione

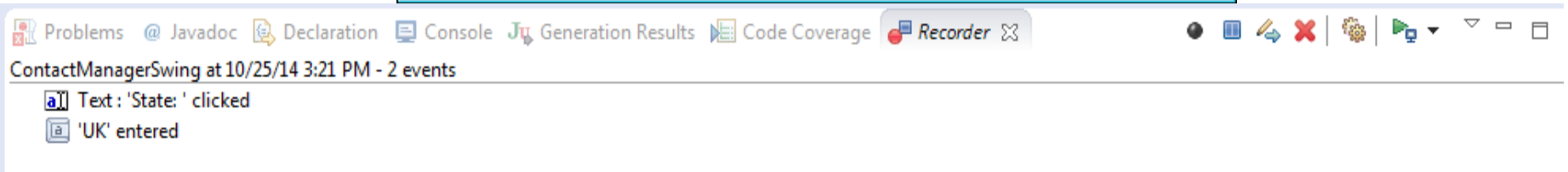




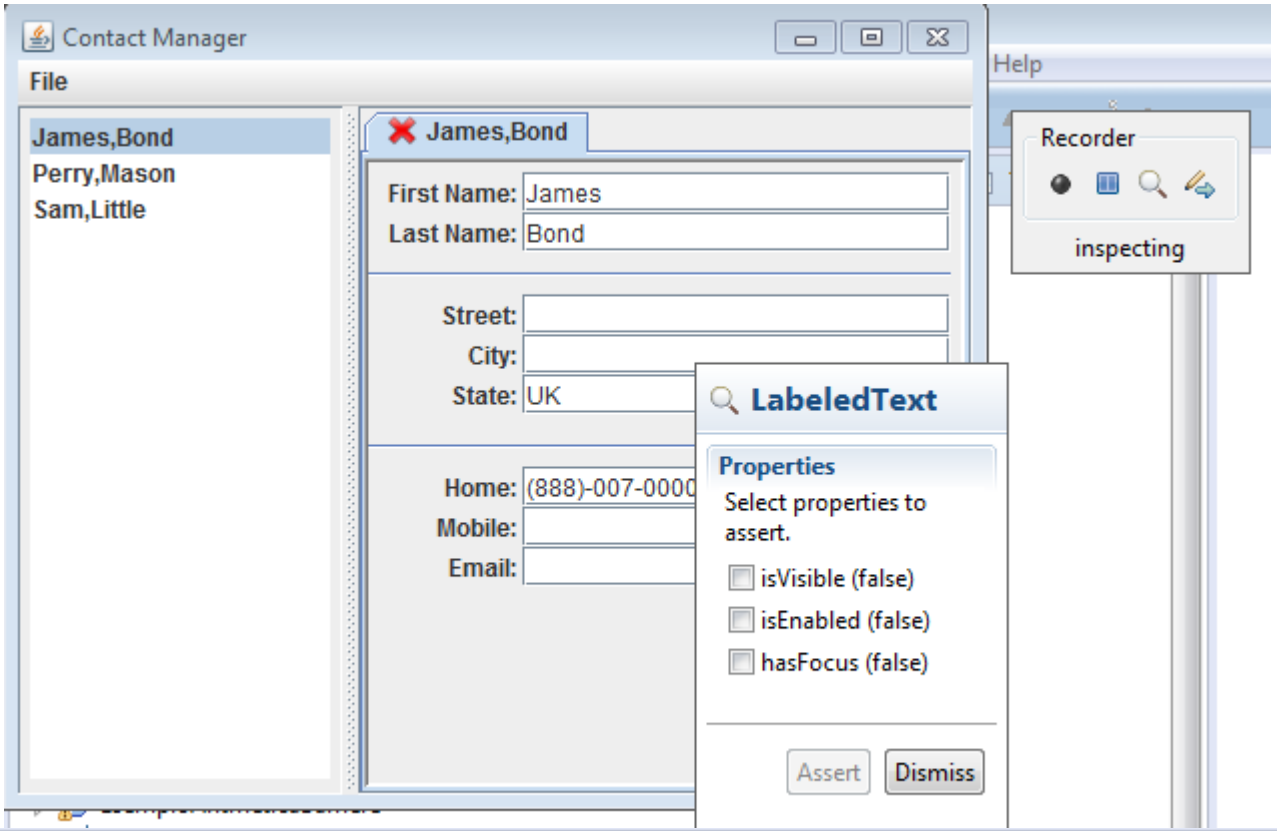
Capture

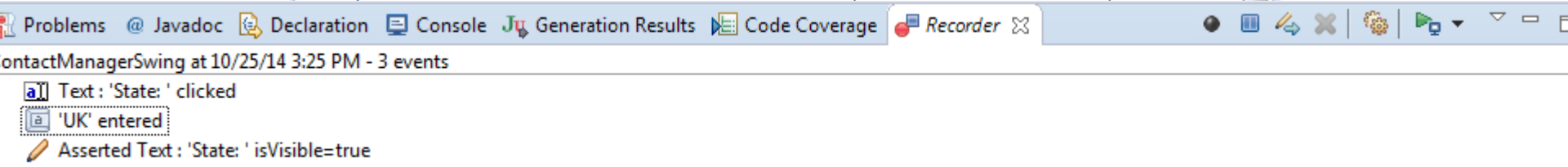

Contact Manager
File
James,Bond
Perry,Mason
Sam,Little
James,Bond
First Name: James
Last Name: Bond
Street:
City:
State: UK
Zip:
Home: (888)-007-0000
Office:
Mobile:
Email:


Recorder
recording

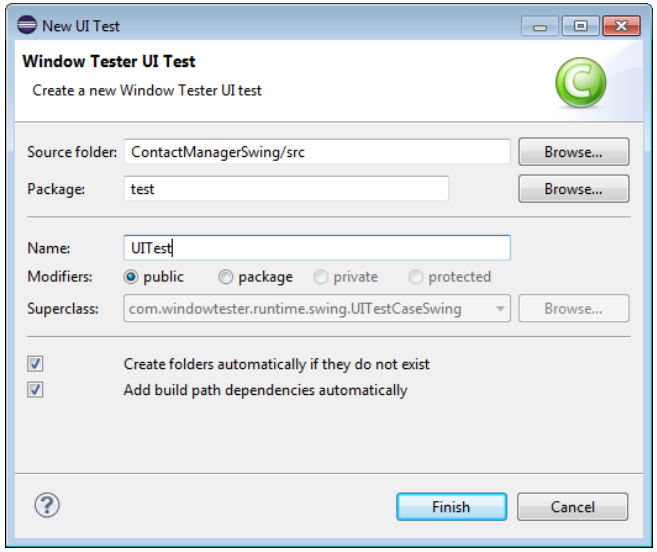

Problems
Javadoc
Declaration
Console
Generation Results
Code Coverage
Recorder
ContactManagerSwing at 10/25/14 3:21 PM - 2 events
Text : 'State: ' clicked
'UK' entered

Esplorazione visuale degli elementi e suggerimento di asserzioni





Generazione automatica di test



- I test possono essere rieseguiti come ogni altro test

```
package test;
import com.windowtester.runtime.swing.locator.LabeledTextLocator;
import com.windowtester.runtime.swing.UITestCaseSwing;
import com.windowtester.runtime.IUIContext;
import com.windowtester.runtime.swing.locator.JMenuItemLocator;

public class UITest extends UITestCaseSwing {
    public UITest() {
      super(contactmanager.ContactManagerSwing.class);
    }

    public void testUI() throws Exception {
      IUIContext ui = getUI();
      ui.click(new LabeledTextLocator("State: "));
      ui.enterText("UK");
      ui.assertThat(new LabeledTextLocator("State: ").isVisible());
      ui.click(new JMenuItemLocator("File/Exit"));
    }
}
```

Alcuni limiti

- **I widget sono individuati in base al nome o all'etichetta che mostrano a video**
  - Il programmatore deve preventivamente evitare di utilizzare widget senza nome
- **Ulteriori asserzioni su altri elementi devono essere aggiunte manualmente**
- **Alcuni elementi ed eventi non sono «visti» (ad esempio all'interno di un File/Open)**
  - Bisogna trovare metodi alternativi in quei casi

```
public void testUITest2() throws
Exception {
IUIContext ui = getUI();
ui.click(new JButtonLocator("6"));
ui.click(new JButtonLocator("*"));
ui.click(new JButtonLocator("7"));
ui.click(new JButtonLocator("="));

IWidgetReference wref =
(IWidgetReference) ui.click(2, new
JTextComponentLocator(JTextArea.class));
JTextArea textArea = (JTextArea)
wref.getWidget();
assertEquals("42.0",textArea.getText());
}
```

Appendice

UISpec4J

- UISpec4J è una libreria a supporto del testing funzionale e di unità di applicazioni Java con interfaccia utente basata su Swing, che fa uso di Junit
- In pratica, mette a disposizione metodi e oggetti che consentono di interrogare direttamente gli elementi dell'interfaccia utente
  - **UISpec4J funziona senza problemi sulle versioni Java fino alle 1.6.25**
- http://www.uispec4j.org/

Test Case in UI Spec

- **Una classe di test UISpec va ad estendere la classe UISpecTestCase**
  - `public class AddressBookTest extends UISpecTestCase`
- **Per poter eseguire i test mantenendo un punto di controllo sull'applicazione sotto test, il metodo setup conterrà :**
  - `protected void setUp() throws Exception { setAdapter(new MainClassAdapter(Main.class, new String[0])); }`
    - **il parametro String[0] rappresenta la linea di comando, in questo caso vuota, della chiamata dell'applicazione**

Esempio: Calcolatrice (1/2)

```
package Calcolator;
import org.uispec4j.*

public class TestWhiteBox extends
   UISpecTestCase {
private Window main;
protected void setUp() throws
   Exception {
setAdapter(new
   MainClassAdapter(Starter.class, new
   String[0]));

main = getMainWindow();
}
```

```
public void testSumOK() throws
   Exception{
Button num1 = main.getButton("4");
num1.click();
Button plus = main.getButton("+");
plus.click();
Button num2 = main.getButton("4");
num2.click();
Button equals =
   main.getButton("=");
equals.click();
assertEquals("8.0",main.getTextBox
   ().getText());
}
```

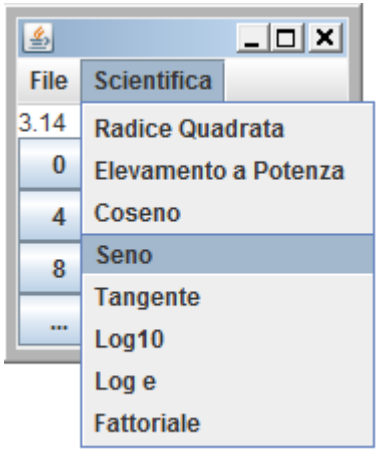
- **La finestra principale è ottenuta col metodo getMainWindow()**
- **I riferimenti ai Button sono ottenuti con getButton(String) in base all'etichetta che essi mostrano**
- **Il riferimento alla TextBox è ottenuto con getTextBox() approfittando della circostanza che essa è unica, in tale interfaccia**

Esempio: Calcolatrice (2/2)

```
package Calcolator;
import org.uispec4j.*

public class TestWhiteBox extends
    UISpecTestCase {
private Window main;
protected void setUp() throws
    Exception {
setAdapter(new
    MainClassAdapter(Starter.class,
    new String[0]));

main = getMainWindow();
}
```




```
public void testSenoNotOK() throws Exception{
Button numc1 = main.getButton("3");
numc1.click();
Button numc2 = main.getButton(".");
numc2.click();
Button numc3 = main.getButton("1");
numc3.click();
Button numc4 = main.getButton("4");
numc4.click();
MenuItem menu=
    main.getMenuBar().getMenu("Scientifica");
menu.click();
MenuItem menu2= menu.getSubMenu("Seno");
menu2.click();
assertEquals("0.0015926529164868282",main.getT
    extBox().getText());
```

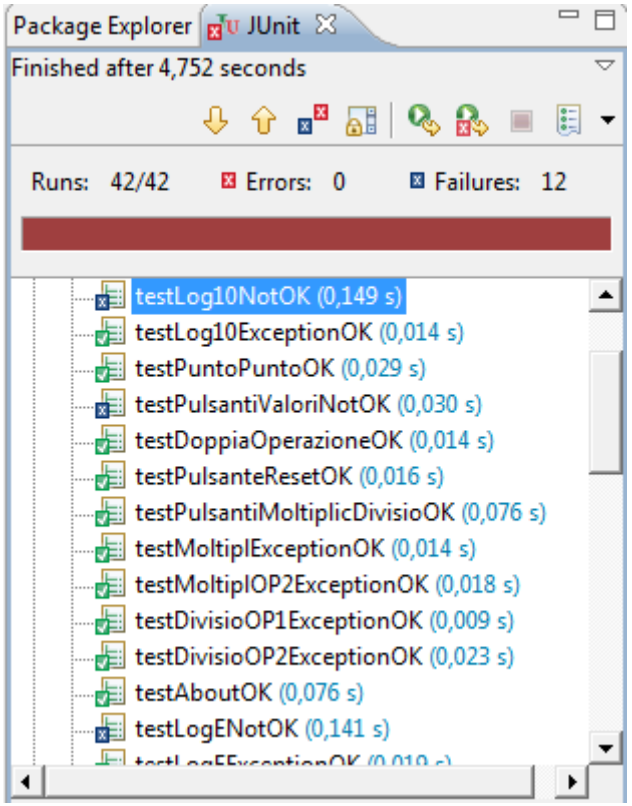
- **Il riferimento al menu è ottenuto con i metodi getMenuBar().getMenu(String), dove la String è l'etichetta visualizzata del Menu**
- **L'asserzione è una asserzione tra stringhe, per cui assertEquals è ben posto; se si fosse trattato di un confronto tra reali si sarebbe dovuto utilizzare un asserzione del tipo:**
  - AssertTrue (Math.abs(valoreAtteso-valoreOttenuto)<epsilon)

Esecuzione dei Test Case

- **L'esecuzione dei test si ottiene sfruttando il framework Junit**
- **Ulteriori metodi possono essere personalizzati per poter individuare gli elementi dell'interfaccia in altri modi**
- **E' possibile interagire anche con i Dialog**

- **Ulteriori informazioni sono disponibili nel tutorial all'indirizzo:**

http://www.uispec4j.org/tutorial



Considerazioni su UISpec4J

- **UISpec4J è uno strumento molto semplice e abbastanza potente, a supporto del testing di interfacce utente Swing (le più diffuse) di applicazioni Java interattive**

- **UISpec4J estende Junit e può essere utilizzato in tutti i processi nei quali viene utilizzato quest'ultimo**
  - Progettazione di casi di test da parte dello sviluppatore, per il testing di unità
  - Progettazione di casi di test per il testing funzionale black box

- **UISpec4J si presta poco alla generazione automatica di casi di test, ma è naturalmente sempre utilizzabile per l'esecuzione automatica e la valutazione automatica dell'esito dei test**

Considerazioni su UISpec4J

- **UISpec4J propone diversi modi per identificare un widget**
  - **Per ID, per titolo, per etichetta visualizzata**
- **Nonostante tali metodi siano molto utili al tester, potrebbero non essere sufficienti**
  - **Ad esempio, widget anonimi non possono essere controllati direttamente**
- **Un software che curi la sua *testabilità* dovrebbe sempre agevolare il testing dei propri componenti permettendone un agevole indirizzamento**
  - **E' buona pratica dare sempre nomi statici ai widget**
  - **Sistemi come Android mettono a disposizione meccanismi basati su XML per obbligare alla dichiarazione statica dei widget**

Approfondimento: Considerazioni su UISpec4J

- **Per gestire i dialog è opportuno gestire dei thread**
- **Nell'esempio, l'interazione successiva con due dialog avviene tramite due click su due pulsanti di due dialogbox**
- **Per l'interazione con un dialog viene istanziato un oggetto process**

```
@Test
   public void testRegolamento() throws Exception {
      try{
         WindowInterceptor.init(new MainClassTrigger(impiccato.class, new String[0]))
         .process(new WindowHandler() {
            public Trigger process(Window dialog) {
               return dialog.getButton("Leggi il regolamento").triggerClick();
               }
            })
         .process(new WindowHandler() {
            public Trigger process(Window dialog) {
                  msg = dialog.getDescription().contains("Lo scopo di");
                  assertTrue(msg);
                  return dialog.getButton("Ok").triggerClick();
                  }
            })
          .run();
         } catch(org.uispec4j.interception.InterceptionError e) {
          if (e.getCause().toString().contains("junit.framework.AssertionFailedError"))
               throw new junit.framework.AssertionFailedError(e.getCause().toString());
          else throw new java.lang.Exception(e);
            }
      }
```

Approfondimento: Considerazioni su UISpec4J

- **In quest'altro esempio, c'è una sequenza di click su button e inserimento di caratteri in dialog successivi**

```
@Test
  public void test1p1() throws Exception {
    try{
      WindowInterceptor.init(new MainClassTrigger(impiccato.class,
new String[0]))
      .process(new WindowHandler() {
        public Trigger process(Window dialog) {
          return dialog.getButton("Inizia a giocare").triggerClick();
          }
        })
      .process(BasicHandler.init().triggerButtonClick("Un giocatore"))
.process(BasicHandler.init().setText("Foo").triggerButtonClick("OK"))
      .process(BasicHandler.init().triggerButtonClick("OK"))
.process(BasicHandler.init().setText("n").triggerButtonClick("OK"))
.process(BasicHandler.init().setText("f").triggerButtonClick("OK"))
.process(BasicHandler.init().setText("o").triggerButtonClick("OK"))
.process(BasicHandler.init().setText("r").triggerButtonClick("OK"))
.process(BasicHandler.init().setText("m").triggerButtonClick("OK"))
.process(BasicHandler.init().setText("a").triggerButtonClick("OK"))
.process(BasicHandler.init().setText("t").triggerButtonClick("OK"))
.process(BasicHandler.init().setText("i").triggerButtonClick("OK"))
.process(BasicHandler.init().setText("c").triggerButtonClick("OK"))
.process(new WindowHandler() {
        public Trigger process(Window dialog) {
          msg = dialog.containsUIComponent(TextBox.class,
"Complimenti, hai vinto!!!").isTrue();
          assertTrue(msg);
          return dialog.getButton("Ok").triggerClick();
          }
        })
.process(BasicHandler.init().triggerButtonClick("Annulla"))
      .run();
      } catch(org.uispec4j.interception.InterceptionError e) {
if(e.getCause().toString().contains("junit.framework.AssertionFailedErro
r"))
          throw new
junit.framework.AssertionFailedError(e.getCause().toString());
        else throw new java.lang.Exception(e);
        }
    }
```

FEST

- Un framework alternativo a UISpec4J
  - **FEST: Fixtures for Easy Software Testing**

- **https://code.google.com/p/fest/**