# Testing Automation

# Testing Automation

- È l'insieme delle tecniche e delle tecnologie che consentono di automatizzare (anche parzialmente) alcune attività del Processo di Testing.

- Alcune aree di intervento:
  - **A) Generazione dei Casi di Test**
  - **B) Preparazione ed Esecuzione del Test**
  - **C) Valutazione dell'esito dei casi di test**

  - **Valutazione dell'efficacia potenziale di Test Suite e tecniche di testing**

# Tecniche di generazione automatica dei casi di test

# A) Tecniche di Generazione dei casi di test

- **Negli approcci al testing presentati finora, si è sempre considerato il task di Test Case Design come un task svolto manualmente dal tester**
  - A causa del numero di casi di test necessari per un testing efficace, l'operazione di progettazione manuale dei casi di test può essere molto onerosa
- **Tecniche per la generazione automatica dei casi di test possono ridurre drasticamente i costi e i tempi legati alla fase di test design**
  - Può però essere necessaria una fase di valutazione dell'efficacia dei casi di test e una fase di riduzione dei casi di test ridondanti

# Generazione automatica dei casi di test

- **I test case possono essere generati automaticamente (alcuni esempi):**
  - Dall'analisi della documentazione di analisi (specifica dei requisiti)
  - Dall'analisi della documentazione di progetto (Model based testing)
  - Dall'analisi statica del codice sorgente
  - Dall'osservazione di esecuzioni reali dell'applicazione (user session testing)
  - Dalla interazione casuale con l'applicazione (Monkey testing)
  - A partire da altri test precedentemente realizzati

# Conoscenza del tipo dell'input

- Come ricavare automaticamente casi di test a partire dalla conoscenza della tipologia dei valori di input?

- Conoscenza del tipo dell'input
  - Possiamo generare casi di test con valori appartenenti a quel tipo
    - Tipo carattere →generiamo casi di test con ogni carattere
    - Tipo booleano →generiamo casi di test con vero e falso
    - Tipo casella a discesa→generiamo casi di test per ognuno dei valori
    - Tipo scelta multipla→generiamo casi di test corrispondenti a tutti gli elementi dell'insieme delle parti
    - Tipo intero →dovremmo generare casi di test corrispondenti a tutti i possibili valori interi rappresentabili …
    - Tipo stringa →dovremmo generare casi di test corrispondenti a tutte le parole possibili (almeno di lunghezza pari a quella massima consentita dal campo di input) …

- Problema: per molti tipi, il numero di possibili casi di test da generare è troppo elevato (rispetto alle risorse da dedicare al testing)

# Conoscenza della semantica dell'input

- Se è nota anche la *semantica* del dato in input è possibile limitare l'insieme dei valori rilevanti
  - Esempi:
    - Mese →generiamo casi di test corrispondenti ai valori tra 1 e 12
    - Velocità →generiamo casi di test corrispondenti a numeri>=0
    - Giorno →generiamo casi di test corrispondenti ai valori tra 1 e 31
    - Voto di Laurea → generiamo casi di test corrispondenti ai valori tra 66° 110 e al valore *110 e lode*
    - Parola italiana →generiamo casi di test per ognuna delle parole presenti su un dizionario italiano

- Problemi:
  - anche in questo caso il quantitativo di casi di test può essere in certi casi troppo elevato
  - E'possibile che l'utente ignori la semantica e immetta valori non previsti

# Dizionario dei dati

- Il caso migliore si ha quando, in fase di specifica dei requisiti, ad ogni dato in input è associato un *dizionario dei dati* con tutti i valori che esso può assumere

- Problemi:
  - Il dizionario dei dati non è sempre facile da definire
    - In un campo di testo libero, l'utente può scrivere qualsiasi valore
  - Lo sforzo necessario alla definizione di un dizionario dei dati per ogni input potrebbe essere superiore a quello che si è deciso di dedicare alle attività di testing …

- **Noti i valori che vogliamo inserire nei campi di input, è poi possibile utilizzare una delle tecniche di testing combinatorio note per generare casi di test eseguibili**

# Model Based Testing

- Generare automaticamente casi di test a partire da un modello di progetto

- Le tecniche di model based testing spesso sono mutuate dalle tecniche di model-checking
  - Differenza: il model checking è una tecnica di verifica formale di correttezza, il model based testing è solo una tecnica di generazione di casi di test

# Model Based Testing

- Esempi di tecniche Model Based:
  - Dato un CFG, cercare di generare casi di test che coprono suoi cammini
  - Dato un FSM, cercare di generare casi di test che eseguono suoi percorsi
  - Dato un workflow diagram di un processo/di un servizio, cercare di generare casi di test che eseguono suoi percorsi
  - Dato un modello Simulink, cercare di coprire le sue possibili modalità di esecuzione
  - ...

# Model Based Testing

- Il caso basilare di Model Based Testing è quando il «modello» dell'applicazione considerato è il suo stesso codice sorgente
- Analizzatori statici del codice sorgente possono capire:
  - L'architettura in termini di package, classi, metodi (e loro parametri), attributi
  - Qualche informazione riguardo la semantica degli input (ad es. alcuni valori cui sono sensibili)
  - Il control flow graph
- A questo punto, essi possono generare casi di test che abbiano lo scopo di coprire (potenzialmente) gli elementi del codice sorgente dell'applicazione (istruzioni, decisioni, etc.)

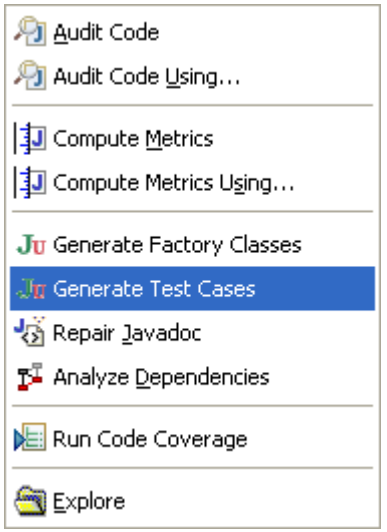
# CodePro Analytix

- **Plug-in multifunzionale per Eclipse offerto da Google**
  - https://developers.google.com/java-dev-tools/codepro/doc/
  - http://googlewebtoolkit.blogspot.it/

  - Scaricabile direttamente da:
    - http://dl.google.com/eclipse/inst/codepro/latest/3.7
    - Aggiornato ad Eclipse Indigo; dovrebbe funzionare anche per le versioni successive
- **Tutorial e documentazione accessibili da:**
  - https://developers.google.com/java-dev-tools/codepro/doc/

# Test Case Generation con CodePro Analytix a partire dal codice sorgente

- **CodePro Analytix fornisce uno strumento per la generazione automatica di test Junit.**
- **Per eseguirlo è sufficiente scegliere l'opzione Generate Test Case dal menu contestuale CodePro Tools**
  - E' possibile chiedere di generare test relativi ad un solo metodo, una sola classe o tutto il progetto
- **I test vengono generati in un nuovo progetto chiamato *nomedelprogetto*Test**
- **I test sono immediatamente eseguibili**



# Tecniche di test supportate

- **Dal menu CodePro/Preferences/Junit è possibile scegliere diverse modalità di generazione:**
  - Junit 3 o 4
  - Due tecniche di generazione
    - *Flow based*
      - **Che genera test tenendo solo conto delle interfacce dei metodi e delle decisioni**
    - *Heuristic based*
      - **Che tiene anche conto delle condizioni, cercando di eseguire test che le provino**
  - Molte altre opzioni di generazione

# Esempio: funzione valida

## Funzione valida

```
public static boolean valida(int d, int
    m, int a) {
if (d<1 || d>31 || m==0 || a<=1582)
return false;
Boolean bisestile= (a%4==0);
if (bisestile && a%100==0 && a%400!=0)
bisestile=false;
if ((m==2 && d>29)||(m==2 && d==29 &&
    !bisestile))
return false;
if ((m==4 || m==6 || m==9 || m==11) &&
    d>30)
return false;
return true;

}
```

## Unico metodo di test generato in modalità flow-based

```
@Test
public void testValida_1()
throws Exception {
int d = 29;
int m = 2;
int a = 1583;

boolean result = Calendario.valida(d, m, a);

// add additional test code here
assertEquals(false, result);
}
```

# Esempio: funzione valida

- **La tecnica heuristic based genera automaticamente 30 casi di test**

```
@Test
public void testValida_1()
throws Exception {
int d = 0;
int m = 0;
int a = 1582;

boolean result = Calendario.valida(d, m, a);

// add additional test code here
assertEquals(false, result);
}
```

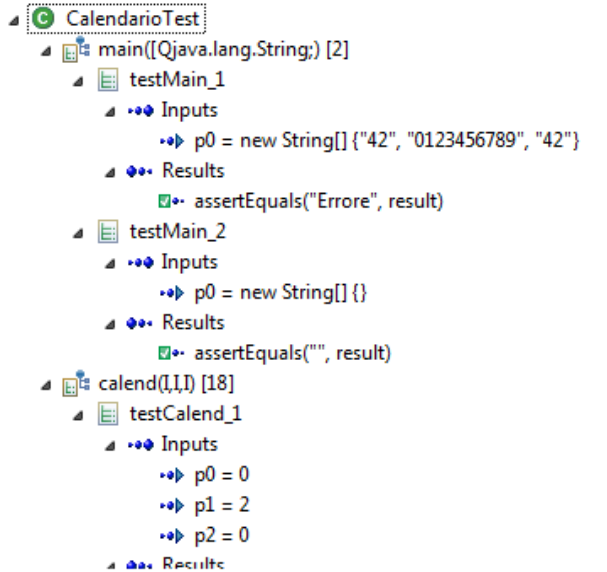
```
@Test
public void testValida_2()
throws Exception {
int d = 1;
int m = 11;
int a = 1583;

boolean result = Calendario.valida(d, m, a);

// add additional test code here
assertEquals(true, result);
}
```

```
@Test
public void testValida_3()
throws Exception {
int d = 29;
int m = 2;
int a = 1583;

boolean result = Calendario.valida(d, m, a);

// add additional test code here
assertEquals(false, result);
}
```

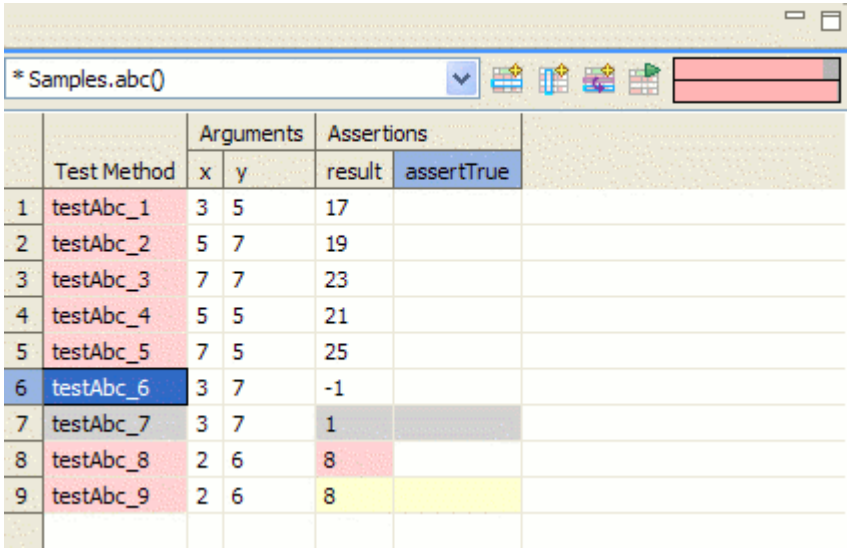
```
@Test
public void testValida_4()
throws Exception {
int d = 30;
int m = 4;
int a = 1583;

boolean result = Calendario.valida(d, m, a);

// add additional test code here
assertEquals(true, result);
}
```

# Cenni ad altre features di Test Generation

- **CodePro è in grado di generare anche Mock secondo il framework EasyMock**
  - Funzionalità sperimentale, tipicamente utilizzata per sostituire classi del package sql
- **Test Case Outline consente una vista veloce di tutti i test case**




- CodePro fornisce anche un Test Editor che consente di vedere i test generati in forma tabellare e apportare automaticamente le modifiche nel codice
  - I colori codificano gli esiti dei test, asserzione per asserzione




| | Test Method | Arguments x | Arguments y | Assertions result | Assertions assertTrue |
|---|---|---|---|---|---|
| 1 | testAbc_1 | 3 | 5 | 17 | |
| 2 | testAbc_2 | 5 | 7 | 19 | |
| 3 | testAbc_3 | 7 | 7 | 23 | |
| 4 | testAbc_4 | 5 | 5 | 21 | |
| 5 | testAbc_5 | 7 | 5 | 25 | |
| 6 | testAbc_6 | 3 | 7 | -1 | |
| 7 | testAbc_7 | 3 | 7 | 1 | |
| 8 | testAbc_8 | 2 | 6 | 8 | |
| 9 | testAbc_9 | 2 | 6 | 8 | |

# Monkey Testing e Random Testing

- Secondo l'**infinite monkey theorem** una scimmia che scrive a macchina battendo tasti a caso in un tempo infinito sarà in grado di scrivere l'Amleto di Shakespeare.

- Nelle tecniche di Random Testing vengono generate sequenze casuali di input allo scopo di testare l'applicazione
  - Il Monkey Testing è la specializzazione del Random Testing al caso di un sistema interattivo, nel quale gli input sono eventi
    - *Non ci sono oracoli dipendenti dai casi test*
    - *Con questa tecnica possono essere trovati soltanto crash oppure possono essere valutate condizioni invarianti di malfunzionamento*
      - **Esempio: regola di usabilità: ci sono pulsanti che escono fuori dall'area dello schermo**
    - *Anche la lunghezza della sequenza può a sua volta essere scelta casualmente*

- **Il Monkey testing si applica essenzialmente a problemi di Robustness Testing**

# Random Testing

- **Il Random Testing ha un'efficienza molto scarsa**
- **Non è molto efficiente ma può condurre alla scoperta di malfunzionamenti che non vengono trovati da altre strategie di testing più «intelligenti»**
- **Viene condotto in maniera totalmente automatica**

# Tipologie di Monkey Testing

- **Dumb Monkey Testing**
  - Gli eventi sono generati in maniera totalmente casuale, secondo una distribuzione uniforme di probabilità
- **Brilliant Monkey Testing**
  - Gli eventi e le sequenze di eventi sono generati secondo una distribuzione di probabilità specifica (spesso dipendente da osservazioni precedenti della distribuzione degli eventi di utenti reali)
- **Smart Monkey Testing**
  - Come nel Brilliant Monkey Testing, ma ulteriori euristiche possono essere introdotte per evitare ad esempio di ripetere eventi o sequenze di eventi già testate
  - Esempi (demo) di un tool di Brilliant Monkey Crash testing dell'Università di Valencia
    - *https://staq.dsic.upv.es/sbauersfeld/Blank.html*
    - *http://thedailycrash.blogspot.de/*

# Monkey Fuzz

- **Un semplice esempio di Monkey in ambiente windows è Monkey Fuzz**

- **http://monkeyfuzz.codeplex.com/**

- **Consente soltanto di generare eventi casuali**
- **Analizzando i log è possibile capire se si sono verificati crash**
  - Attenzione: può causare interazioni non volute con il sistema

# Monkey

**Monkey è un'utility interna fornita con l'android SDK, che è in grado di generare eventi utente pseudocasuali su una qualsiasi interfaccia, registrando gli eventuali crash**

- Monkey gira all'interno del dispositivo; per avviarla bisogna passare per adb. Ad esempio, da linea di comando:

```
adb shell monkey -v -p
  com.porfirio.orariprocida2011 30
```

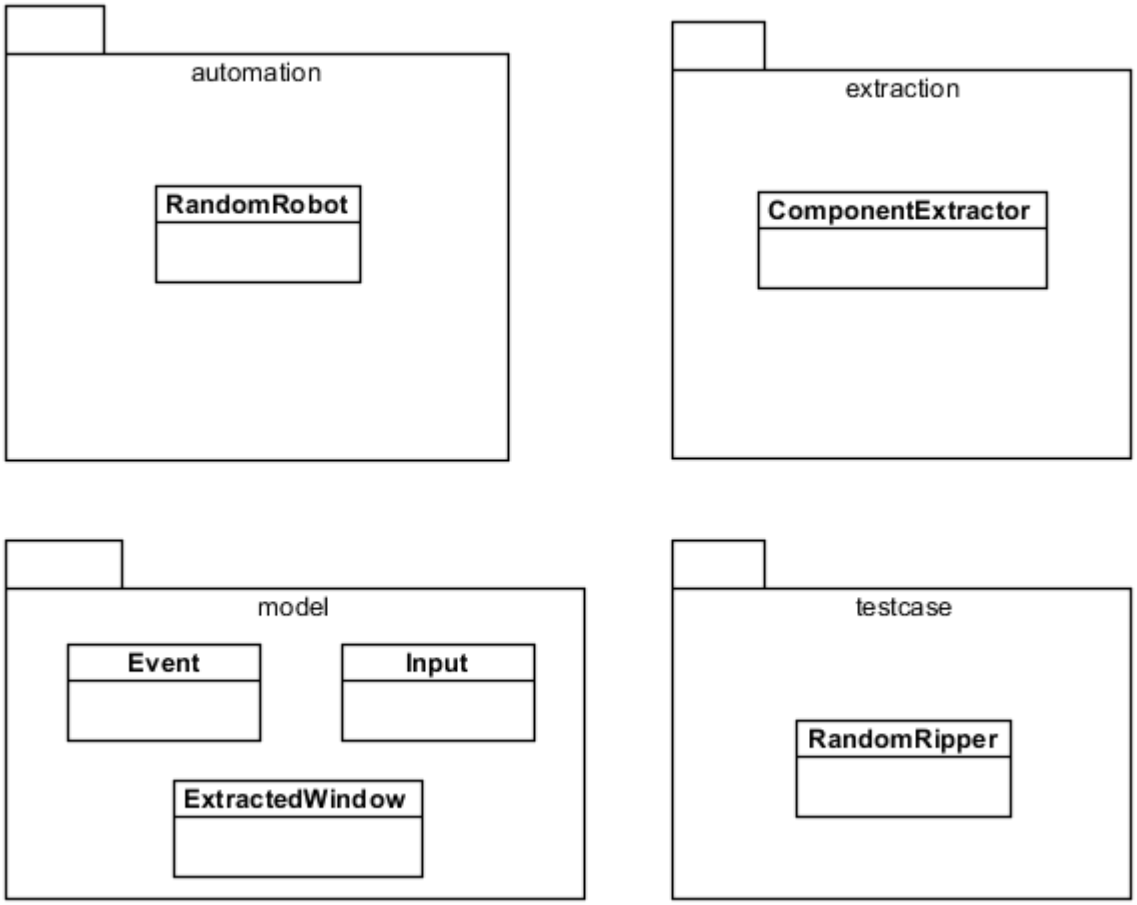
# Output di Monkey

```
:Monkey: seed=0 count=30
:AllowPackage: com.porfirio.orariprocida2011
:IncludeCategory: android.intent.category.LAUNCHER
:IncludeCategory: android.intent.category.MONKEY
// Event percentages:
//   0: 15.0%
//   1: 10.0%
//   2: 15.0%
//   3: 25.0%
//   4: 15.0%
//   5: 2.0%
//   6: 2.0%
//   7: 1.0%
//   8: 15.0%
:Switch:
      #Intent;action=android.intent.action.MAIN;category=android.intent.category.LAUNCHER;launchFlags=0x10000000
      ;component=com.porfirio.orariprocida2011/.OrariProcida2011Activity;end
    // Allowing start of Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER]
      cmp=com.porfirio.orariprocida2011/.OrariProcida2011Activity } in package com.porfirio.orariprocida2011
:Sending Pointer ACTION_MOVE x=-4.0 y=2.0
:Sending Pointer ACTION_UP x=0.0 y=0.0
:Sending Pointer ACTION_DOWN x=47.0 y=122.0
Events injected: 30
:Dropped: keys=0 pointers=0 trackballs=0 flips=0
## Network stats: elapsed time=7766ms (7766ms mobile, 0ms wifi, 0ms not connecte
d)
// Monkey finished
```

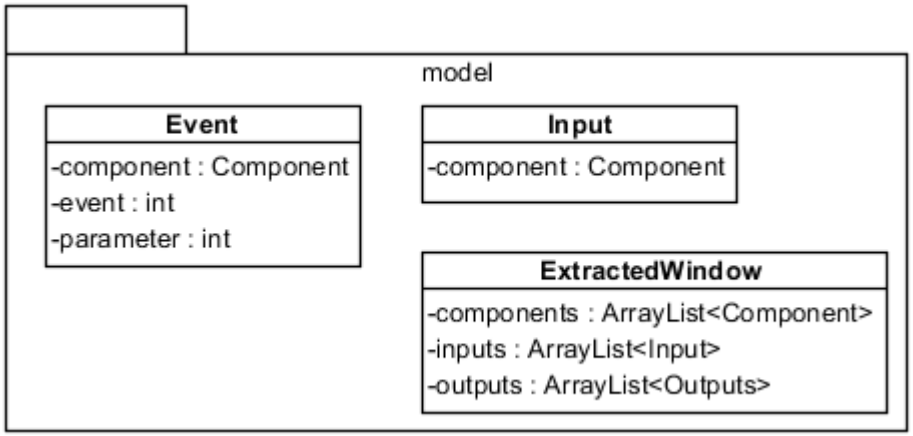
# Random GUI Java Tester

- **Uno strumento di testing random generato dal gruppo di Ingegneria del Software della Università Federico II**
- **Genera test random consistenti in sequenze di eventi su interfacce grafiche AWT e Swing**
  - Le GUI sono analizzate grazie alle proprietà di reflection e alla libreria FEST
  - Può essere eseguito sia da eclipse che standalone, sia da main che come caso di test, eventualmente integrato con uno strumento di misura della coverage
  - Può essere anche aggiunto come libreria ad un progetto esistente, con l'aggiunta di un semplice punto d'avvio

# Random GUI Java Tester



# Package Model




- **ExtractedWindow**
  - Contiene le informazioni relative ai componenti della Window
- **Input**
  - Astrae un Input, destinato ad un componente
- **Event**
  - Astrae un Evento, destinato ad un componente

# Package Extractor

- **ComponentExtractor**
  - È la classe che si occupa di recuperare informazioni della Window mostrata a schermo
    - *Le informazioni vengono restituite come un'istanza della classe ExtractedWindow*
      - **ArrayList<Component> contiene un riferimento agli oggetti Component che costituiscono la Window**
    - *In particolare, distingueremo tra elementi su cui eseguiremo degli eventi ed eventi sui quali inseriremo anche dei valori in input*
    - *Dopo aver recuperato le informazioni, la classe genera un elenco di eventi che possono essere scatenati sulla Window*
    - *Tali eventi, insieme ai Component identificati come campi di Input, sono aggiornati nell'istanza della classe ExtractedWindow*

# Package Extractor

- **Ad esempio, sul campo di testo (Text Field) inseriremo dei valori (quindi è un Input), mentre sulla CheckBox ci limiteremo a eseguire un evento, cioè il click (codificato come RIPPER_EVENT_CLICK)**

```
if (component instanceof javax.swing.JTextField) {
    Input input = new Input();
    input.component = component;
    ret.addInput(input);
}

if (component instanceof javax.swing.JCheckBox) {
    Event evt = new Event();
    evt.component = component;
    evt.event = Event.RIPPER_EVENT_CLICK;
    ret.addEvent(evt);
}
```

# Package Automation

- **RandomRobot**
  - È la classe che si occupa dell'esecuzione degli eventi e della valorizzazione dei campi di input
  - Le strategie per scegliere quali eventi eseguire e quali valori (eventualmente casuali) inserire vengono scritte qui
  - L'evento inviato al Robot viene tradotto in interazioni con la GUI
    - *Vengono utilizzate le librerie FEST e la classe Robot di Java*
      - **https://github.com/alexruiz/fest-swing-1.x**
      - **https://docs.oracle.com/javase/7/docs/api/java/awt/Robot.html**

# Package Automation

- **Ad esempio:**

```
if (input.component instanceof javax.swing.JTextField) {
        JTextComponentFixture fixture = new
JTextComponentFixture(robot, (JTextField)input.component);
        fixture.setText( RANDOM.getRandomIntString(100) );
}
```

- **Se il componente trovato sulla GUI è un'istanza di un campo di testo (TextField), allora creiamo un oggetto di interazione con questo componente (fixture) e su di esso eseguiamo un metodo setText, che ha come parametro un numero casuale tra 0 e 100**
  - Questo valore casuale sarà chiaramente il valore messo in input nel campo di testo
  - Diverse strategie potrebbero tenere conto di classi di equivalenza, valori limite, etc.

# Package TestCase

- **RandomRipper**
  - È un caso di test jUnit
  - Contiene il *main loop* del Java Random Ripper
    - *Per ogni evento casuale da eseguire, analizza la window cercando componenti, sceglie un evento da eseguire, vi associa eventuali inputs e lo esegue (Fire)*

```
for(numeroDiEventi)
    extractedWindow = extractor.ExtractWindow()
    event = scheduler.getRandomEvent(extractedWindow)
    inputs = extractedWindow.getInputs()
    if (event != null)
        fireRandomEvent(event, inputs)
```

# Caratteristiche del random testing

- **Fattori in grado di influenzare l'efficacia del random testing:**
  - Numero di eventi provati
  - Lunghezza dei casi di test provati
    - *E' stato provato che test troppo brevi sono poco efficaci poiché non riescono a coprire alcune condizioni (ad esempio un contatore che arriva al valore massimo), mentre test troppo lunghi rendono poco probabile la copertura di condizioni che si verificano solo a inizio test (ad esempio codice legato al primo inserimento di qualche dato)*
      - **http://crest.cs.ucl.ac.uk/fileadmin/crest/sebasepaper/Arcuri09e.pdf**
  - Ampiezza dell'insieme dei possibili valori di input
  - …

# Terminazione del Random testing

- **Quando terminare il random testing?**
  - La condizione più soddisfacente per terminare il testing è il raggiungimento della copertura massima raggiungibile (*saturazione*)
    - Ma potremmo non essere in grado di riconoscerla!
      - **La copertura a saturazione è l'insieme di tutte le righe di codice escluse quelle non raggiungibili, che non sono però note a prio**

# Due tecniche per determinare la terminazione di un random testing

- **Eseguire n sessioni di test random in parallelo**
  - Tecnica 1) Terminare i test quando tutte le sessioni hanno raggiunto lo stesso identico insieme di copertura
  - Tecnica 2) Terminare i test quando il codice raggiunto dalla sessione con la maggiore copertura include il codice raggiunto da ognuna delle altre sessioni

# Sperimentazione

- **Queste tecniche sono attualmente oggetto di sperimentazioni**
  - Per trovare il valore ottimale del numero di sessioni (finora risultati soddisfacenti si sono ottenuti con n>=8)
  - Per capire quale codice potrebbe essere coperto ma non viene coperto al tempo in cui si verifica la condizione di terminazione
    - *Esempio: vittoria in un gioco di strategia*
  - Per capire quale delle due condizioni di terminazione è più efficiente

# Caratteristiche del Random Testing

- Il Random Testing può portare ad esecuzioni diverse da quelle progettate da un tester, quindi a scoprire nuovi difetti
- Il Random Testing può portare a rieseguire moltissime volte la stessa esecuzione
  - Un Random Testing meno stupido può avere memoria delle esecuzioni precedenti, per non ripeterle
- Il Random Testing può essere eseguito in parallelo
- I Test Random non hanno oracolo. Possono essere utilizzati solo per:
  - Cercare possibili situazioni di crash / eccezioni;
  - Verificare la violazione di proprietà invarianti

# Proposta di progetto

- **Implementare tecniche di testing random alternative evolvendo lo strumento Random Ripper**
  - Valutare la bontà di queste tecniche su di un insieme preesistente di piccole e medie applicazioni java con interfaccia grafica
  - Eventualmente, reimplementando uno strumento di random testing con Maveryx

# Proposta di progetto

- **Implementare uno strumento che possa eseguire test in parallelo, eventualmente su più macchine, reali o virtuali e possa confrontare dinamicamente i risultati di copertura ottenuti**

# Proposta di progetto

**Trovare e confrontare diverse tecniche di terminazione del testing random**

# Risorse sul Random Testing


- Andrea arcuri. Random Testing. http://www.uio.no/studier/emner/matnat/ifi/INF4290/v10/undervisningsmateriale/INF4290-RandomTesting.pdf
- J. W. Duran and S. C. Ntafos. An evaluation of random testing. IEEE Transactions on Software Engineering, 10(4):438–444, 1984.
- S. C. Ntafos. On comparisons of random, partition, and proportional partition testing. IEEE Transactions on Software Engineering, 27(10):949–960, 2001.
- Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05). ACM, New York, NY, USA, 213-223. http://cm.bell-labs.com/who/god/public_psfiles/pldi2005.pdf

# Search Based Software Testing

- E’ una branca del software testing nella quale il testing è visto come un problema di ottimizzazione (analogo a quelli trattati dalla ricerca operativa), sui quali vengono applicate tecniche euristiche come ad esempio:
  - Algoritmi genetici
  - Simulazione della tempra
  - Tabu search
  - Programmazione lineare

# Algoritmi genetici applicati al testing

- Per applicare un algoritmo genetico ad un problema di testing è necessario:
  - Modellare i casi di test (esemplare) in forma di sequenze (gli elementi vengono detti alleli)
    - *ad esempio come sequenze di valori di input o come sequenze di eventi*
  - Proporre operatori di crossover
    - *Che incrociano due esemplari (casi di test) generando altri due esemplari*
  - Proporre operatori di mutazione
    - *Che variando in maniera casuale uno degli alleli, generano un nuovo esemplare*
  - Proporre una metrica che valuti la bontà di un test case (fitness locale)
    - *Ad esempio, la quantità di codice che copre oppure la quantità di difetti che trova*
  - Proporre una metrica che valuti la bontà complessiva di una test suite (fitnell globale)
    - *Ad esempio la quantità totale di codice coperta dalla test suite oppure il totale dei difetti scoperti*

# Algoritmi genetici applicati al testing

- Gli algoritmi genetici sono iterativi. Data una test suite iniziale, ad ogni iterazione
  - Si creano nuovi test case della test suite effettuando dei crossover
    - *Vincolo: i nuovi test ottenuti devono essere eseguibili*
  - Si creano nuovi test modificando valore in input dei test esistenti
  - Si valuta la fitness locale di tutti I test
  - Si selezionano I test case con la migliore fitness, tali da mantenere lo stesso quantitativo di test case dell'iterazione precedente
  - Si calcola la fitness globale
  - Si termina se si è arrivati ad un valore di fitness globale considerato soddisfacente secondo un criterio prescelto
- Numerose varianti si possono ottenere facendo variare frequenza e tipologia degli operatori, regole di selezione, misure di fitness, criterio di terminazione, etc.
- L'esperienza mostra come queste tecniche portino, più o meno velocemente, a scoprire test suite con buoni valori di fitness globale

# EvoSuite


- **EvoSuite è uno strumento per la generazione di test di unità compatibili con Junit per applicazioni Java**
- **Può essere scaricato e utilizzato sia in forma standalone che come plug-in di Eclipse o IntelliJIdea**
  - Gli autori lo hanno testato su sistemi Linux e Mac, ma affermano che dovrebbe funzionare anche sotto Windows
    - *http://www.evosuite.org/downloads/*
- **EvoSuite è un software prodotto da ricercatori universitari (inizialmente dell'università di Birmingham) ma supportato anche da Google**

# EvoSuite in poche parole

- **Genera casi di test basandosi su informazioni ottenute tramite analisi statica (come CodePro, ad esempio)**
- **Prova questi casi di test valutandone la capacità di coprire il codice**
- **Iterativamente, nell'ambito di una tecnica genetica:**
  - Applica tecniche di mutazione e crossover dei test per generare nuovi casi di test
  - Genera mutazioni dei programmi
    - *Le mutazioni sono generate secondo le regole implementate nel programma muJava*
  - Valuta la capacità dei casi di test di scoprire i programmi mutanti
    - *Per scoprire un mutante si confronta l'output ottenuto col mutante con quello ottenuto sul programma originale*
  - Ottimizza la test suite ottenuta preferendo i test che sono riusciti
    - *A coprire più codice*
    - *A coprire codice che non hanno coperto gli altri casi di test*
    - *A scoprire mutanti*

# Ulteriori caratteristiche

- **EvoSuite è una tecnica euristica**
  - Cerca di generare mutazioni del programma che somiglino ai possibili errori dei programmatori
  - Adotta una tecnica euristica per decidere quando è il momento di terminare
- **EvoSuite è completamente automatica**
- **EvoSuite genera test Junit completamente riusabili**

Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software.

# Proposta di progetto

- **Studiare approfonditamente le caratteristiche di EvoSuite**
  - Dal sito http://www.evosuite.org/
- **Provare EvoSuite su programmi diversi da quelli testati e valutarne criticamente i risultati (ad esempio confrontandoli con altre tecniche di test)**
- **Provare ad estendere EvoSuite**
  - Un tutorial è disponibile all'indirizzo http://www.evosuite.org/documentation/tutorial-part-4/
  - Un precedente progetto a disposizione può rappresentare un valido punto di partenza per quest'attività
    - *EstensioneEvosuite_AmalfitanoPacia.rar*

# Testing basato su Sessioni Utente

- **Idea:**
  - E se si utilizzassero i reali utilizzatori di un'applicazione (alpha tester o beta tester) come cavie da osservare per generare casi di test?
- **Vincoli:**
  - L'applicazione deve essere stata già rilasciata
- **Vantaggi:**
  - Lo sforzo per la generazione dei casi di test è molto limitato (supponendo che gli utilizzatori siano volontari e non facciano parte dell'organizzazione …)
  - Reali utilizzatori possono sollecitare il sistema in modi che i progettisti non avevano immaginato e previsto
  - La riesecuzione dei casi di test può essere completamente automatizzata
- **Svantaggi:**
  - Scarsa efficienza (molti test identici tra loro)
  - Ridotta efficacia (molti casi limite non vengono testati)
    - *Non garantiscono contro gli utilizzi maliziosi del software*

# Problemi legati allo User Session Testing

- **Il numero di sessioni da prendere in considerazione al fine di poter avere un insieme significativo di casi di test può essere molto elevato**
  - Molte sessioni possono essere identiche o contenere parti simili
  - Si possono applicare tecniche di minimizzazione per la riduzione del numero di casi di test
- **Il sistema di logging deve essere in grado di monitorare il più possibile anche gli elementi legati all'ambiente di esecuzione**
- **Per poter ottenere esecuzioni significative può essere necessario raccogliere sessioni per molto tempo**
  - Per ridurre tale tempo, si possono utilizzare tecniche di ricombinazione (mutazione) dei test
- **Difficoltà nel riprodurre le reali condizioni di utilizzo dell'applicazione prima del suo rilascio reale**
- **Può essere difficile riutilizzare i casi di test prodotti, a seguito di un intervento di manutenzione sul software**

# Tecniche di riduzione

- **Per ridurre la dimensione di una test suite si può utilizzare una tecnica di minimizzazione basata sulla copertura**
  - Si fissa un obiettivo di copertura
    - *Ad esempio copertura di istruzioni, decisioni, pagine visitate, scenari di casi d'uso …*
  - Si valuta il grado di copertura di ogni caso di test
  - Si esegue un algoritmo in grado di estrarre il più piccolo insieme di casi di test che massimizzi la copertura

- **E' una tecnica analoga a quella utilizzata per trovare l'insieme di implicanti minimo che coprisse tutti i mintermini di una funzione booleana**

# Riduzione dei test cases

**Talvolta, si impongono delle tecniche di riduzione per minimizzare il numero di test cases**

- Esempio: Selezione di un insieme di casi di test per applicazioni Web che siano in grado di eseguire, almeno una volta, ogni pagina Web
  - *Si disegna la matrice di copertura (User Session x Pagine)*

| | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ |
|---|---|---|---|---|---|---|
| US1 | | 1 | 1 | 1 | | |
| US2 | 1 | | 1 | 1 | | |
| US3 | 1 | 1 | 1 | | | |
| US4 | | | | | | 1 |
| US5 | | | | 1 | | 1 |
| US6 | | | | | 1 | |

Un elemento (I,j) vale 1 se la Pagina Pj è eseguita nell'ambito della User Sesson USi

# Tecnica di Riduzione

**Tre criteri:**

**Criterio di essenzialità:**

- Una US è essenziale se è l'unica a coprire una pagina P

**Criterio di dominanza per le righe:**

- USi è dominata da USj se tutte le pagine coperte da USi sono coperte anche da USj

**Criterio di dominanza per le colonne:**

- Pi domina Pj se Pi è contenuta in tutte le US contenenti Pj.

| | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ |
|---|---|---|---|---|---|---|
| US1 | | 1 | 1 | 1 | | |
| US2 | 1 | | 1 | 1 | | |
| US3 | 1 | 1 | 1 | | | |
| US4 | | | | | | 1 |
| US5 | | | | 1 | | 1 |
| US6 | | | | | 1 | |

**US6 è essenziale poichè è l'unica che contiene P5**

**US4 è dominata da US5 poichè tutte le pagine contenute in US4 sono anche contenute in US5**

**P3 domina P1 e P2 poichè P3 è contenuta in tutte le sessioni contenenti P1 e in tutte quelle contenenti P2**

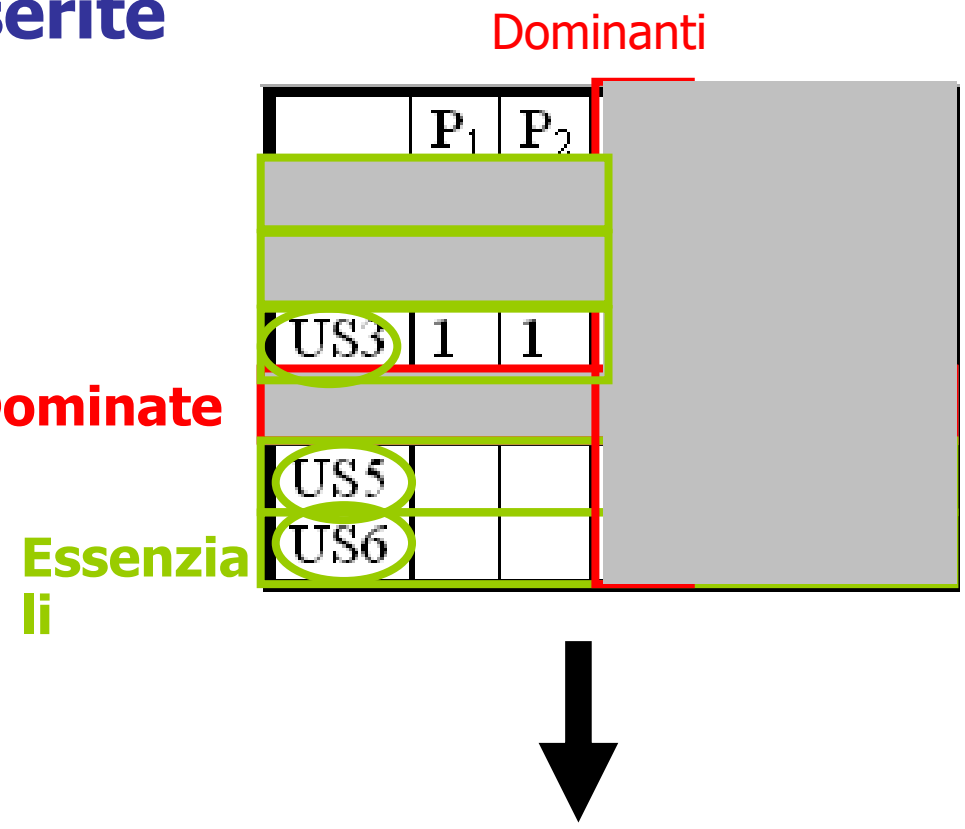
# Applicazione della tecnica di riduzione

**Le US essenziali sono inserite nell'insieme ridotto**

**Le pagine coperte da US essenziali sono eliminate**

**Le US dominate sono eliminate**

**Le Pagine dominanti sono eliminate**



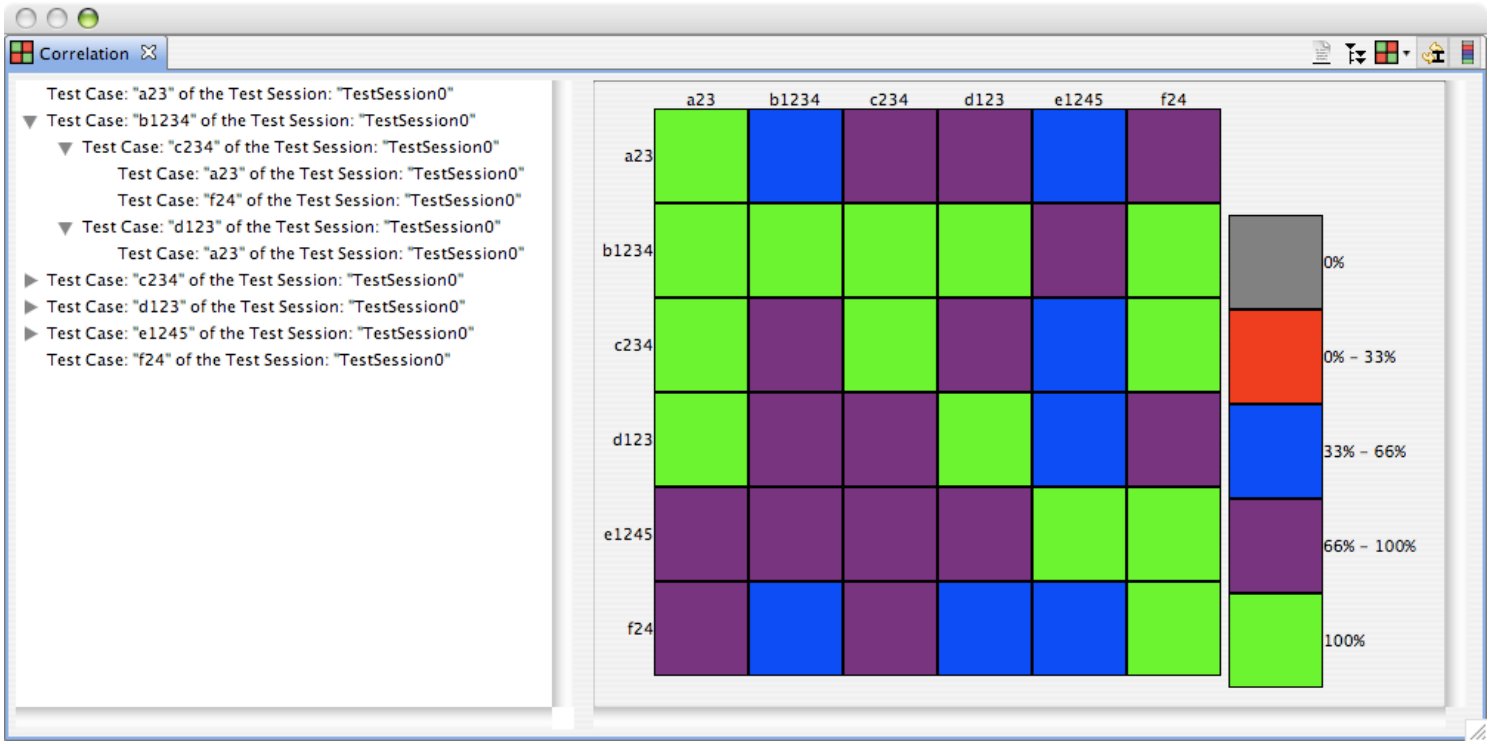

Insieme ridotto:
US3, US5, US6

# Altre tecniche di riduzione

- **Tecniche euristiche basate sulla misura della somiglianza tra casi di test**
- **Ipotesi:**
  - Test "simili" hanno probabilmente lo stesso esito
- **Obiettivo:**
  - Massimizzare la diversità tra i test
    - *Abbiamo bisogno di una tecnica per misurare la somiglianza (o la diversità)*

# Altre tecniche di riduzione

- **Esempio:**
  - Misurare la somiglianza in termini di linee di codice coperte



# Test Case Prioritization

- **Le tecniche di riduzione rientrano nella più ampia famiglia di tecniche di test case prioritization**
  - Dare un peso (euristico) ai casi di test in modo da selezionare il sottoinsieme di casi di test che massimizza questo peso
- **Obiettivo comune di queste tecniche è sempre una massimizzazione dell'efficienza a parità (o con una perdità accettabile) di efficacia**
- **Slide su test case prioritization http://www.cs.umd.edu/~atif/Teaching/Fall2004/StudentSlides/xun1.pdf**

# Test Case Prioritization

- **Bisogna trovare delle regole euristiche in base alle quali misurare l'efficacia potenziale dei test case**
  - Si ordinano i test per efficacia potenziale decrescente
  - Si esegue un sottoinsieme di tali test che soddisfi un criterio di terminazione

# Test Case Prioritization

- **Esempi di metriche euristiche**
  - I test case su parti di codice recentemente modificate sono più efficaci
  - I test case che coprono più codice per primi
    - *La code coverage può essere misurata pesando la diversità di codice coperto*
  - I test case più «veloci» per primi
  - I test case meno «costosi» per primi
  - …
  - Euristiche basate sul giudizio di esperti
  - Random prioritization

# Un ulteriore Approccio: il Testing Mutazionale

- Il **Testing Mutazionale** è una tecnica per la generazione di casi di test.
  - A partire da un sottoinsieme di casi di test, si applicano alcuni **operatori di mutazione** che vadano a modificare/incrociare i dati dei test case esistenti, in modo da ottenere nuovi test case.
  - Es. Si cambia il segno degli input, si raddoppiano i valori di input, si combinano sequenze di input in nuove sequenze, etc…

# Testing Mutazionale

- **Con tale tecnica si possono ottenere Test Suites**
  - più piccole (meno test cases)
  - con maggiore copertura
  - con uno sforzo minore

  rispetto a quelle ottenute semplicemente collezionando sessioni utente
- **Bisogna però eliminare tutti i test cases che risultano inapplicabili.**
- **Questa tecnica è spesso utilizzata per il testing di protocolli.**

# Operatori di mutazione

- **Alcuni esempi:**
  - Modificare l'ordine degli eventi
  - Fondere due test
  - Creare un nuovo test con la prima parte proveniente da un test e la seconda da un'altra
  - Aggiungere dei tempi d'attesa prima di una certa operazione del test
  - …

# Tecniche di esecuzione automatica dei casi di test

**"I am rarely happier than when spending entire day programming my computer to perform automatically a task that it would otherwise take me a good ten seconds to do by hand."**

*- Douglas Adams - From "Last Chance to See"*

# Esecuzione automatica dei casi di test

- **Si tratta della parte più «meccanica» della testing automation**
- **Il completo automatismo si può ottenere scrivendo il codice di test sotto forma di codice eseguibile**
  - **E' possibile ricorrere a linguaggi ed ambienti diversi da quello dove è in esecuzione il software da testare**
    - **Esempio: script di shell che eseguono in maniera batch i software da testare**
      - **Problemi: valido solo per testing black box, poiché non è possibile interagire internamente col software testato**
  - **E' possibile modificare il software sotto test creando punti di esecuzione alternativi**
    - **Ad esempio ulteriori metodi main**
      - **Problemi: il software testato viene così modificato, con il rischio di introdurre ulteriori bug e con la difficoltà di rimuovere il codice di test senza rischiare di dover testare nuovamente il software dopo questa rimozione**

# Esecuzione automatica dei casi di test

- La soluzione più efficace è quella di scrivere codice con framework come Xunit
  - Vantaggi:
    - possiamo eseguire sia test black box che white box, monitorando eventualmente anche lo stato interno del software;
    - le classi di test sono separate da quelle originali, cosicchè non c'è alcun rischio a rimuoverle e non possono influenzare il corretto funzionamento del software
  - Vincolo:
    - Il software deve essere scritto in un linguaggio che supporti la reflection
      - La maggior parte dei moderni linguaggi object-oriented supportano la reflection
        - C non supporta nativamente la reflection

# Tecniche di valutazione automatica dell'esito dei casi di test

# C) Valutazione dell'esito dei casi di test

- **Per poter valutare automaticamente l'esito di un caso di test, esso dovrebbe essere stato oggettivamente definito e un metodo per la sua valutazione deve essere disponibile**
  - Ad esempio, nel caso degli assert in un test Junit
- **In alcuni casi particolari, l'esito di un test non ha bisogno di essere definito, o può essere definito automaticamente**
  - Crash o exception testing
  - Regression Testing
- **Lo stato dell'arte complessivo riguardo il problema della definizione automatica degli oracoli può essere trovato in:**
  - http://mcminn.io/publications/tr3.pdf

# Crash Testing

- **Testare un software in cerca di eccezioni o errori a run-time che interrompano l'esecuzione**
  - Non è necessario definire alcun oracolo: esso corrisponde alla semplice terminazione regolare del caso di test

- **Smoke testing**
  - Una varietà del crash testing, nella quale l'applicazione viene esplorata e navigata il più possibile, cercando di causare un crash
  - Tipicamente, può essere eseguito durante il naturale ciclo di sviluppo dell'applicazione
    - *Ad esempio, un ciclo di smoke testing può essere eseguito durante la notte*
    - *Originariamente utilizzato per il testing di componenti hardware, è molto diffuso nell'ambito dei cicli di sviluppo agili, nei quali una versione integrata e testabile del software dovrebbe essere molto spesso disponibile*

# Testing di regressione

- Si applica in seguito ad un intervento di manutenzione su di un software esistente, per il quale esiste già un piano di test
- Un problema: quali test devono essere riprogettati? E quali test possono essere riusati?
  - Sicuramente devono essere riprogettati tutti i casi di test relativi alla nuova funzionalità implementata (o alla funzionalità modificata)
- Quali altri test dovranno essere rieseguiti?
  - Per determinare quali test preesistenti devono essere rieseguiti, occorre valutare quali altre funzionalità potrebbero essere state influenzate dalla modifica realizzata, ossia eseguire l'Impact Analysis:

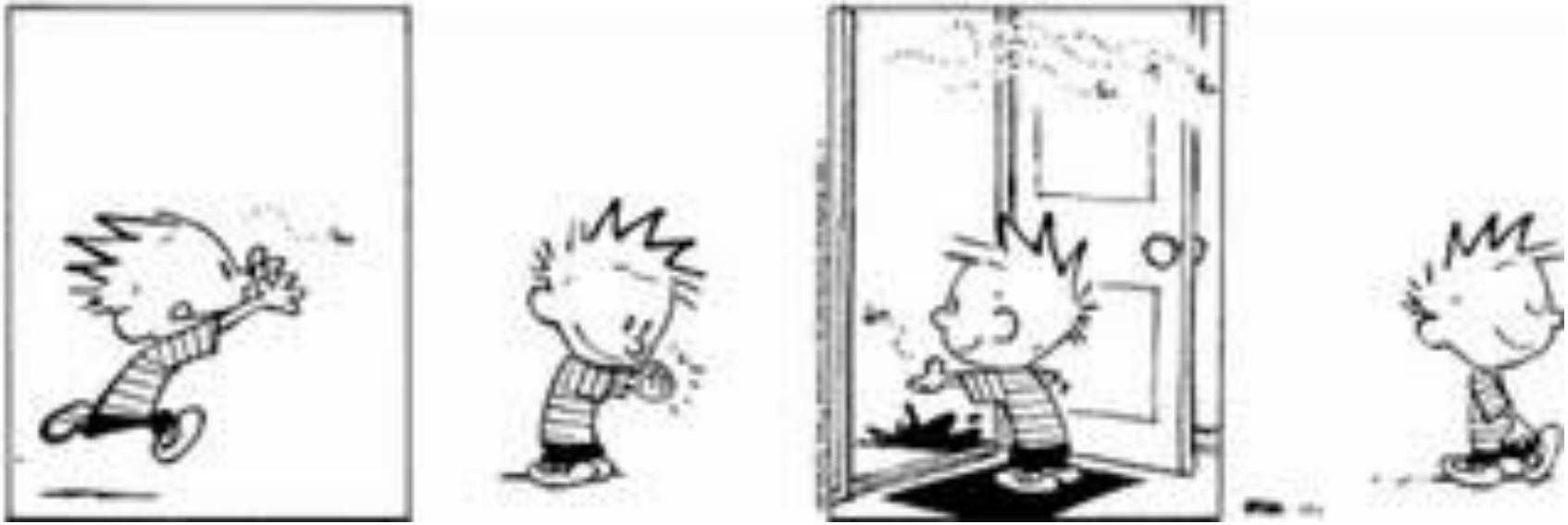
**Quale sarà stato l'*impatto* della modifica sul sistema?**

# Testing di regressione e Ripple effect

- Dopo un intervento di manutenzione, è probabile che la modifica effettuata influisca sul resto del sistema, generando nuovi difetti (è il cosiddetto *ripple effect*).
  - Chi corregge potrebbe non avere una adeguata conoscenza di tutto il sistema e delle sue connessioni
  - Il sistema può regredire ("invecchiare") verso uno stato più difettoso

- Occorre eseguire il Testing di Regressione
  - Particolarmente indicato qualora i testing siano stati definiti in modo da poter essere rieseguiti automaticamente

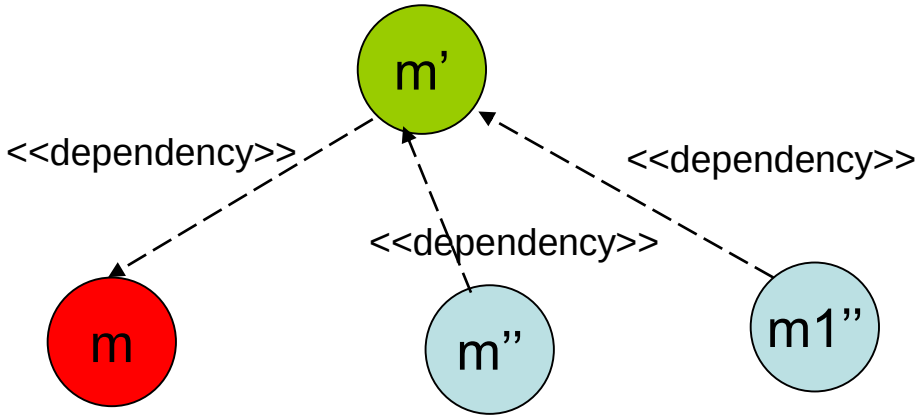
Regression:
"when you fix one bug, you introduce several newer bugs."



# Impact Analysis e grafo delle dipendenze

- L'analisi di impatto è la disciplina che permette di conoscere, data una modifica, quali parti del software possono esserne influenzate (e quindi devono essere ri-testate)
- Una tecnica semplice per la valutazione dell'impatto è basata sul Grafo delle Dipendenze
- Data una modifica su di un modulo m
  - tutti i moduli m' che da essi dipendono (per i quali esista un arco m'→m) sono sicuramente impattati dalla modifica di m
  - Tutti i moduli m" che dipendono da uno qualunque dei moduli m' saranno a loro volta impattati, e così via
- **I casi di test relativi ai moduli impattati (oppure tutti i moduli, nel caso in cui non sia stato possibile effettuare impact analysis) devono essere rieseguiti**
  - L'oracolo del testing di regressione è fornito dall'esito dei test che si otteneva prima di eseguire la modifica

# Grafo delle dipendenze ed Analisi dell'Impatto




Se m è stato modificato, occorrerà controllare (e ritestare) tutti i moduli che dipendono da m (direttamente, come m', ed indirettamente, come m'' e m1'')

Se in fase di progettazione del software, tutte le dipendenze fra artifatti fossero registrate esplicitamente, si potrebbe facilmente eseguire tale Analisi di Impatto

[A.R. Fasolino, G. Visaggio "Improving Software Comprehension through an Automated Dependency Tracer", IEEE Workshop on Program Comprehension,1999]

# Appendice

# Selenium

- Consideriamo il **framework Selenium**, a supporto del testing di interfacce utente di applicazioni Web
  - http://selenium.openqa.org/

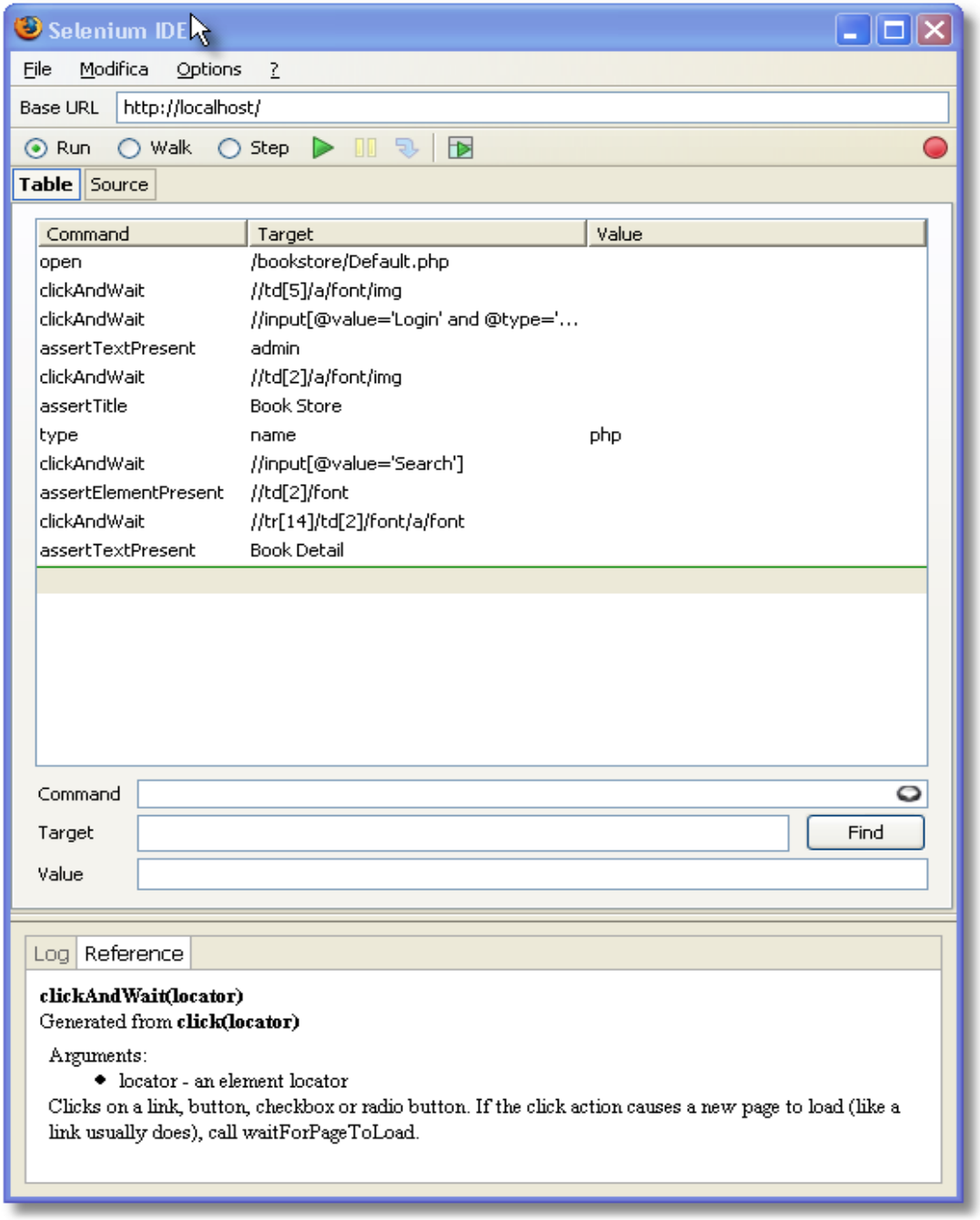
- **Selenium** offre quattro modalità di utilizzo:
  - Selenium IDE
  - Selenium Core
  - Selenium Remote Control
  - Selenium Grid

# Selenium IDE

- **Si tratta di un'estensione di un browser che consente di:**
  - catturare le interazioni tra un utente e una applicazione web (fase di *Capture*)
  - "suggerire" asserzioni relative alla presenza di widget sull'interfaccia utente
  - replicare l'esecuzione di casi di test, mantenendo un log degli esiti dei test (fase di *Replay*)

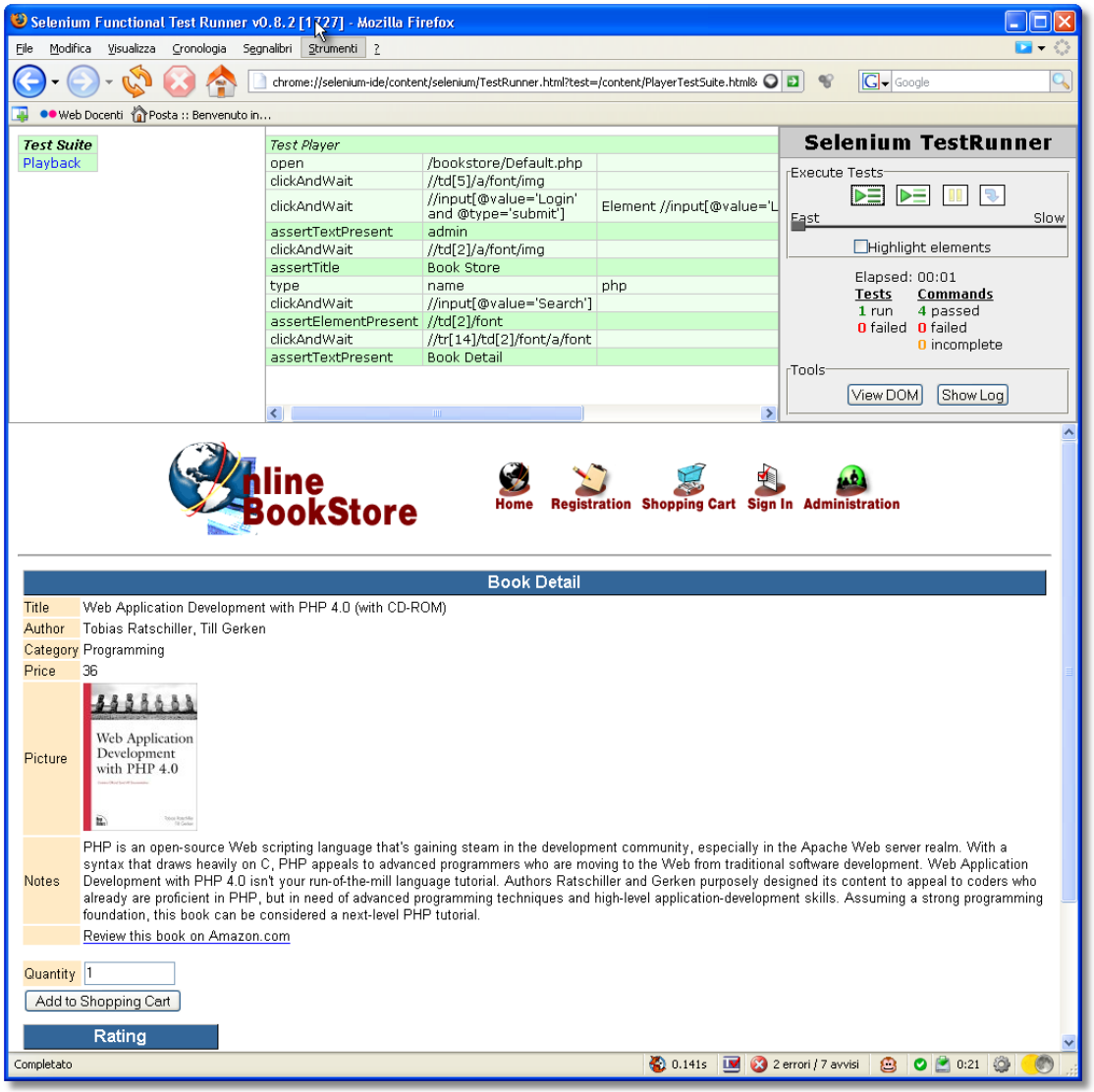
- Selenium è dunque usabile per progettare TC anche a prescindere da un modello formalizzato (es. FSM) della UI.
- Utile per l'esecuzione di Testing di Accettazione

In fase di capture, Selenium IDE mantiene un log delle operazioni effettuate dall'utente e delle asserzioni da egli proposte



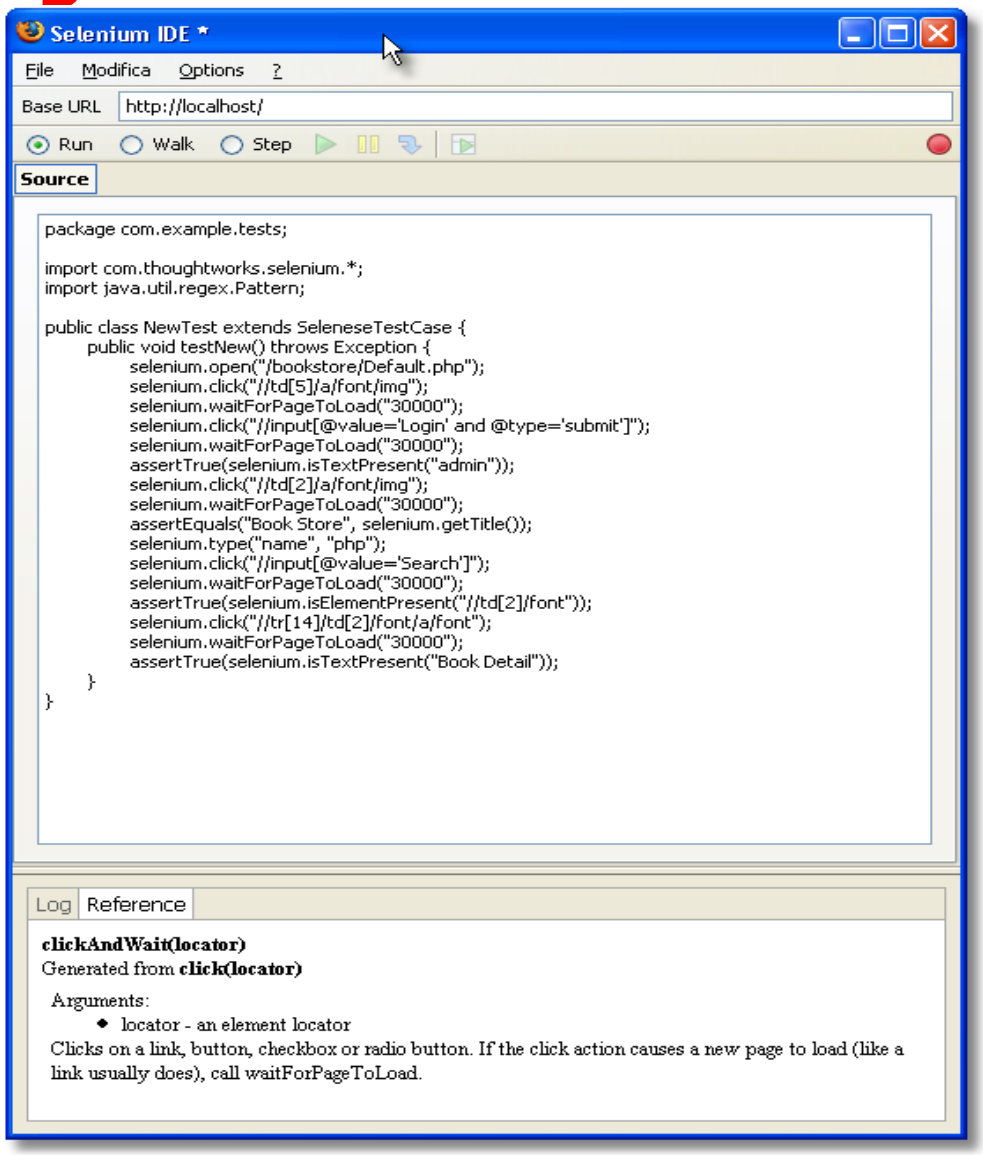
# Replay

In fase di replay, Selenium IDE esegue automaticamente test generati in fase di capture, mantenendo statistiche sul numero di test terminati con successo e falliti



# Codice generato

- In fase di capture, Selenium IDE genera anche del codice sorgente (a scelta in Java, C#, Perl, PHP, Python o Ruby) che può essere eseguito indipendentemente da Selenium IDE
- Il codice generato necessita, per essere eseguito, di packages forniti con Selenium (che formano il Selenium Core)



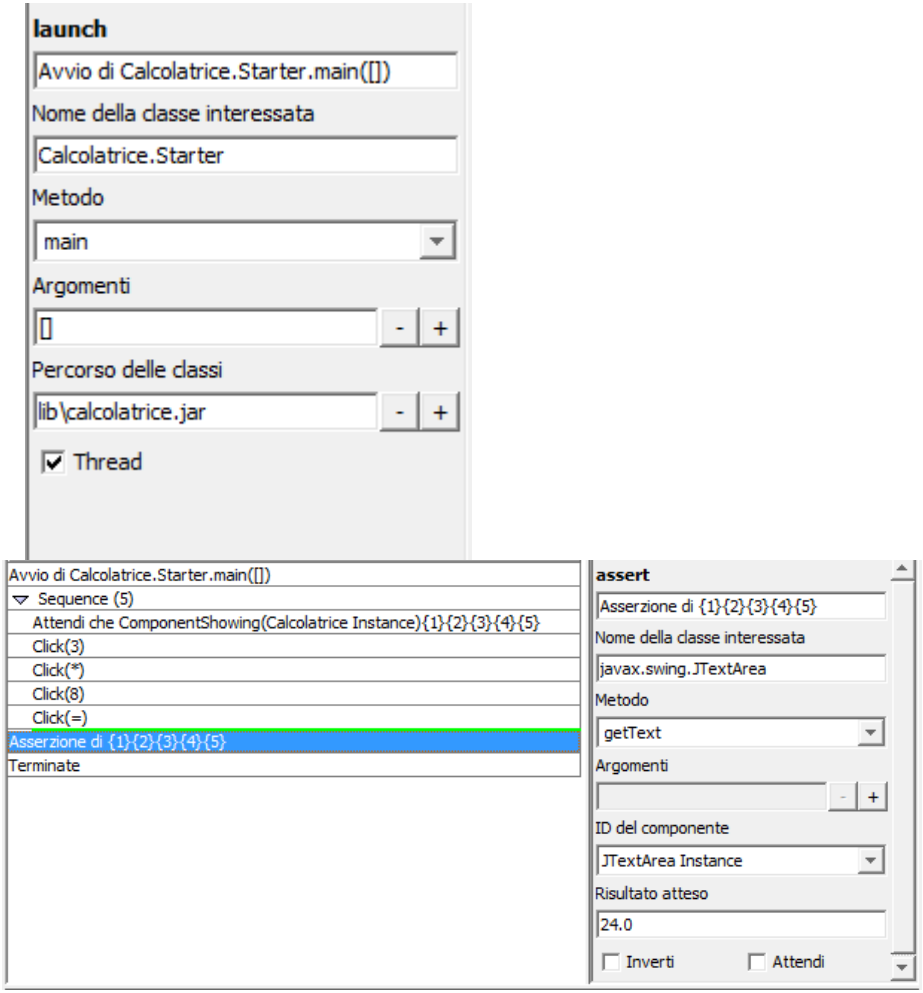
# Abbot e Costello

- Abbot e Costello sono un'altra coppia di tool che aiuta nella programmazione di casi di test per interfacce utente Java (sia AWT che Swing).
  - **Abbot è un insieme di librerie a supporto dell'esecuzione dei test case realizzati**
    - ***Abbot può essere utilizzato in maniera analoga a UISpec4J***
  - **Costello è uno strumento interattivo che fornisce feature per il capture, l'editing, l'esecuzione, la visualizzazione dei risultati dei test ed altro**

**http://abbot.sourceforge.net/doc/overview.shtml**

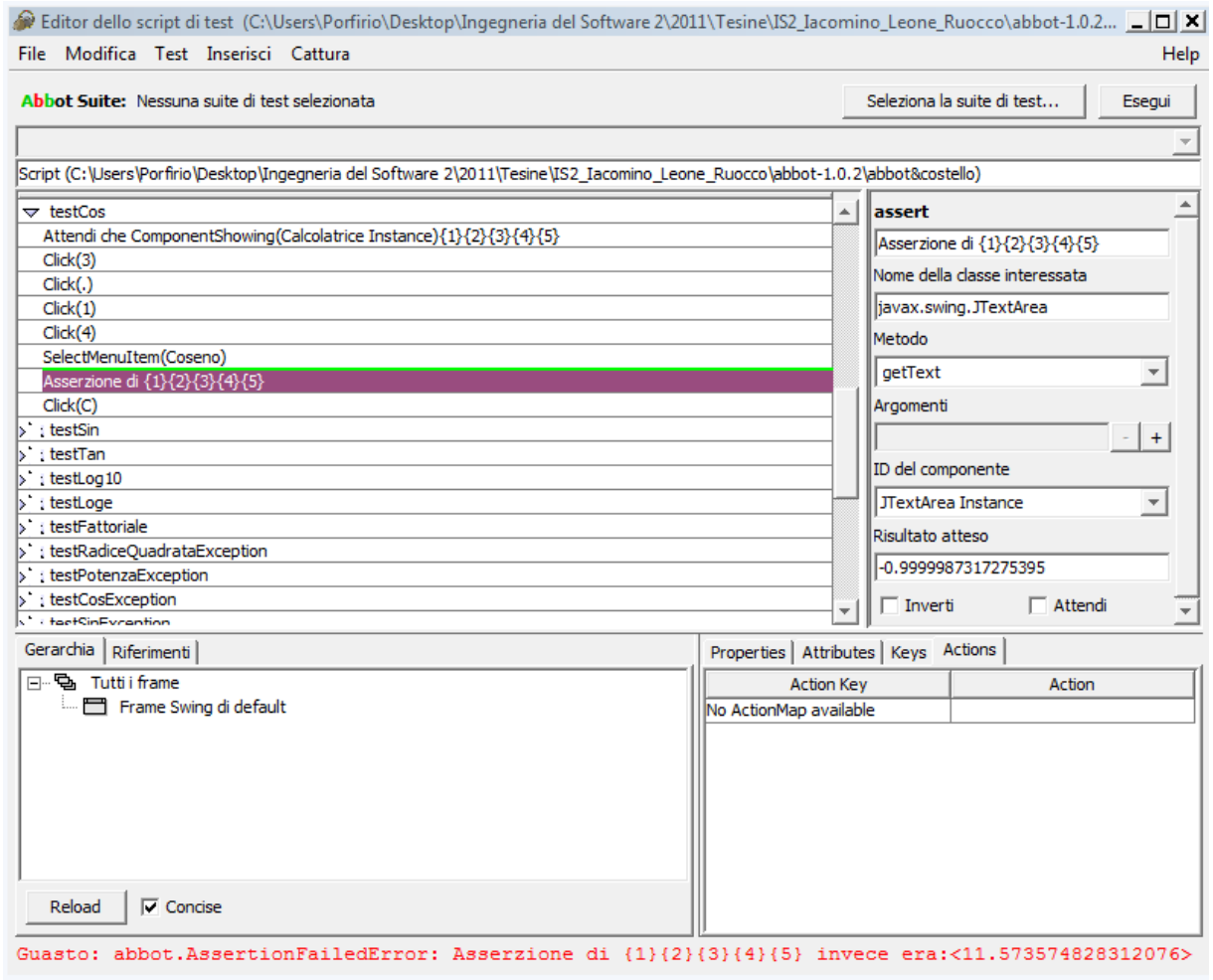
# Uno scenario di utilizzo di Costello

- **Creazione di un nuovo caso di test**
  1. File/New Script:
  2. Imposta classe e metodo di partenza e posizione del jar
  3. Cattura/All Actions
  4. Esegui un esempio di esecuzione sull'applicazione da testare
  5. Premi Shift+F1 dopo aver posizionato il puntatore sul campo da usare per l'asserzione
  6. Imposta un'asserzione
  7. Termina
  8. File/Save (in formato XML)



# Uno scenario di utilizzo di Costello

- **Esecuzione di un caso di test**
  1. File/New Script
  2. Seleziona una test suite
  3. Esegui
  4. Verifica l'esito delle asserzioni



# Utilizzo di Abbot

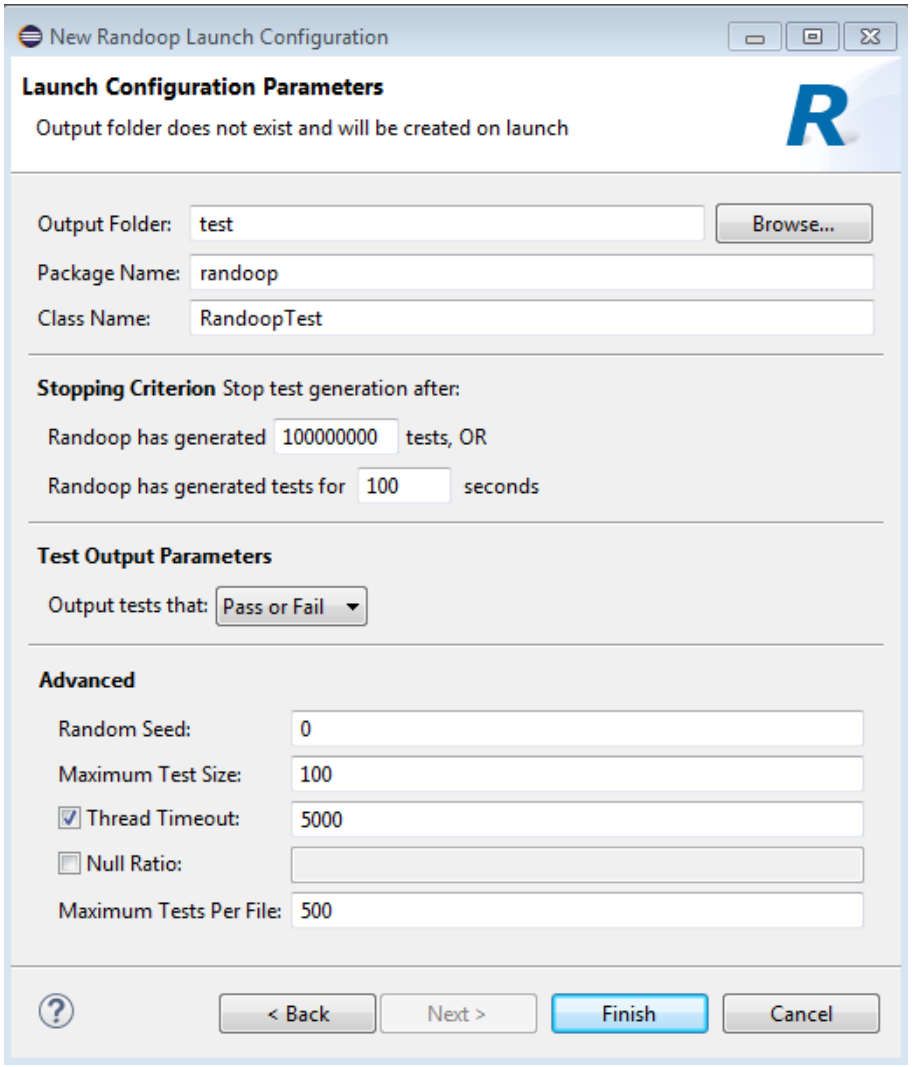
- **Abbot da solo può essere utilizzato in maniera simile ad UISpec4J, per scrivere test, in particolare anche test di unità di singoli componenti:**

```
// Suppose MyComponent has a text field and a button...
MyComponent comp = new MyComponent();
// Display a frame containing the given component
showFrame(comp);

JTextField textField = (JTextField)getFinder().
    find(new ClassMatcher(JTextField.class));
JButton button = (JButton)getFinder().find(new Matcher() {
    public boolean matches(Component c) {
        // Add as much information as needed to distinguish the component
        return c instanceof JButton && ((JButton)c).getText().equals("OK");
    }
});
JTextComponentTester tester = new JTextComponentTester();
tester.actionEnterText(textField, "This text is typed in the text field");
tester.actionClick(button);
// Perform some tests on the state of your UI or model
assertEquals("Wrong button tooltip", "Click here to accept", button.getToolTipText());
```

# Randoop

- **Tecnica per la generazione casuale di casi di test**
  - A partire da una analisi del codice (limitata in particolare a metodi e loro parametri)
  - I test sono costituiti da sequenze casuali di chiamate di metodi su oggetti
  - I test sono generati come script Junit, quindi possono essere rieseguiti
  - Il risultato originale del test è codificato come asserzione. Utilizzabili per test di regressione
  - http://mernst.github.io/randoop/
- **Può essere utilizzato per generare test per un metodo**
  - (o per tutti i metodi di una classe o di un package)
- **Può essere eseguito standalone oppure tramite eclipse**
  - Istruzioni per eclipse: https://rawgit.com/mernst/randoop/master/plugin/doc/index.html
- **Può essere utilizzato sia in presenza del codice sorgente che avendo a disposizione soltanto il bytecode**

# Parametri di Randoop



# Analisi Mutazionale

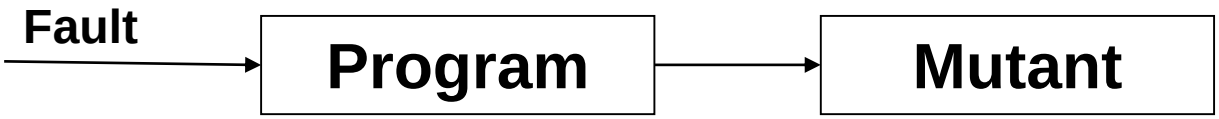
# Analisi Mutazionale

- Un criterio di copertura "ideale" (massima efficacia e massima efficienza) sarebbe quello di riuscire a coprire tutti gli elementi difettosi presenti in un'applicazione.
  - **Se conoscessimo a priori i difetti di una applicazione, potremmo cercare di costruire una test suite che massimizzi sia efficacia che efficienza**
  - **Più realisticamente, possiamo confrontare l'efficacia di diverse test suite tra loro**
- Più in generale, è possibile confrontare la capacità potenziale di rilevazione dei difetti di diverse tecniche di generazione di casi di test
  - L'unico modo per conoscere a priori i difetti di un software consiste nell'iniettarli appositamente (in un software supposto corretto, per ipotesi)

# Analisi Mutazionale

- **Il primo passo consiste nell'immaginare quali possano essere i possibili errori (fault model)**

- **E' necessario proporre un modello degli errori e dei corrispondenti operatori di mutazione**
  - Un operatore di mutazione introduce in un programma, supposto corretto, un difetto (realizzando un'operazione di *fault injection*), trasformando il programma originale in un mutante



# Analisi Mutazionale

- **I difetti (fault) sono inseriti automaticamente nei programmi, ottenendo dei *mutanti***
- **Su ogni mutante generato si vanno ad eseguire i test case della test suite da valutare**
  - L'oracolo per l'esecuzione di tali test è dato dall'output che veniva generato dal programma originale
  - Se l'esito del test è positivo (l'output del mutante differisce da quello del programma originale), allora si dice che il mutante è stato ucciso (*killed)*
  - Altrimenti, il mutante non è stato rivelato dal test, ed è sopravvissuto.
    - *I mutanti (ad esempio quelli che causano errori in compilazione) sono detti triviali quando possono essere scoperti da qualunque caso di test. Questi mutanti vengono subito scartati*
  - **Più mutanti sono uccisi, maggiore fiducia si può avere nella capacità della Test Suite di scoprire difetti!**

# Analisi Mutazionale

**L'efficacia di una test suite si può misurare come**

TER = # killed Mutants / # Mutants

**In conclusione:**

- Una test suite che riesca a rivelare il maggior numero possibile di mutanti è da considerarsi più promettente nella rivelazione di potenziali difetti nell'applicazione
- Una tecnica di generazione generante test suites efficaci nell'uccisione dei mutanti è da considerarsi più promettente nel mondo reale
  - *Nell'ipotesi che i difetti iniettati sono rappresentativi di quelli reali*

# Problemi dell'analisi mutazionale

- **Difficoltà nella modellazione dei difetti di un sistema software**
  - Di solito vengono proposti degli **operatori di mutazione**
    - *Basandosi sull'esperienza generica di chi propone il testing mutazionale riguardo le possibili cause di errore*
    - *Basandosi su di un'analisi statistica dei difetti rilevati in altri software*
  - Gli operatori proposti sono di solito estremamente **generici**, per poter essere applicabili ad ogni software, indipendentemente dal linguaggio adottato e dalle caratteristiche della singola applicazione
    - *Esempi:*
      - **Operatore che sostituisce un'operazione aritmetica con un'altra**
      - **Operatore che sostituisce un operatore relazionale con un altro**
      - **Operatore che inverte la posizione di due righe di programma, etc.**
    - *Con questi operatori, l'iniezione di difetti può essere svolta automaticamente*
    - *Difetti più complessi, legati alla logica dell'applicazione, devono essere iniettati manualmente e rimangono significativi solo per quella specifica applicazione*

# Problemi dell'analisi mutazionale

- **Il numero di possibili difetti è estremamente elevato, già per un piccolo frammento di software**
  - Il numero di mutanti generati da un piccolo insieme di operatori è estremamente elevato
    - *Per ogni mutante dovrebbero essere eseguiti tutti i casi di test di una test suite*
      - **L'approccio è possibile solo in presenza di un ambiente per la testing automation, e anche in questo caso è molto oneroso**
      - **Spesso si decide di applicare gli operatori di mutazione solo "a campione"**

# Frequenza degli operatori di mutazione

- **Bisogna stabilire anche con quale frequenza applicare i distinti operatori di mutazione.**
- **Due tecniche vengono proposte:**
  - Tecnica statistica: si osservano le occorrenze di difetti in applicazioni reali e li si inietta nelle stesse proporzioni
  - Tecnica 'fair': si analizza il codice, contando tutte le opportunità di iniezione delle diverse tipologie di difetti. I difetti vengono iniettati seguendo queste proporzioni.

# Problemi dell'analisi mutazionale

**Altri problemi:**

- Non tutti i difetti di un sistema software sono legati ad errori nel codice sorgente
  - *Si pensi a difetti del tipo di: mancanza di un requisito, scorretta interpretazione di un requisito, etc.*
- Alcuni difetti portano ad errori sintattici e sono già rivelati dal compilatore
- Non è possibile proporre operatori generali che riproducano gli errori semantici

# Vantaggi

- L'analisi mutazionale è una tecnica completamente automatica, che può essere eseguita in batch senza l'assistenza del tester.
  - *Occorre però disporre di strumenti automatici per la generazione dei mutanti, l'esecuzione dei test, la valutazione dei risultati (…)*

- Si rivela utile come banco di prova per il confronto in ambiente sperimentale tra diverse tecniche di generazione di test suite, per valutare quale sia in grado di produrre test in grado di rilevare più difetti.