Verifica e Validazione del Software

Testing White Box

Riferimenti

- Ian Sommerville, Ingegneria del Software, capitoli 22-23-24 (più dettagliato sui processi)
- Pressman, Principi di Ingegneria del Software, 5° edizione, Capitoli 15-16
- Ghezzi, Jazazeri, Mandrioli, Ingegneria del Software, 2° edizione, Capitolo 6 (più dettagliato sulle tecniche)

Testing Strutturale (White Box)

- Il Testing White Box è un testing strutturale, poichè utilizza la struttura interna del programma per ricavare i dati di test.

- **Tramite il testing White Box si possono formulare criteri di copertura più precisi di quelli formulabili con testing Black Box**

  - Test White Box che hanno successo possono fornire maggiori indicazioni al debugger sulla posizione dell'errore

Criteri di Copertura

- Fondate sull'adozione di metodi di Copertura degli oggetti che compongono la struttura dei programmi:
- *istruzioni – strutture controllo – flusso di controllo -…*
- definizione di un insieme di casi di test (input data) in modo tale che gli oggetti di una definita classe (es. istruzioni, archi del CFG, predicati, strutture di controllo, etc.) siano attivati (coperti) almeno una volta nell'esecuzione dei casi di test

Criteri di Copertura e relative Misure di Test Effectiveness

- Criteri di selezione
- **Copertura dei comandi (statement test)**
- **Copertura delle decisioni (branch test)**
- **Copertura delle condizioni (condition test)**
- **Copertura delle decisioni e delle condizioni**
- **Copertura dei cammini (path test)**
- **Copertura dei cammini indipendenti**

- Criteri di adeguatezza
- **n.ro comandi eseguiti/ n.ro comandi eseguibili**
- **n.ro archi percorsi/ n.ro archi percorribili**
- **n.ro cammini percorsi/ n.ro cammini percorribili**
- **n.ro cammini indip. percorsi/n.ro ciclomatico**

UN MODELLO DI RAPPRESENTAZIONE DEI PROGRAMMI: il Control-Flow Graph

- Il grafo del flusso di controllo (Control-Flow Graph) di un programma P:
  - **CFG (P) = <N, AC, nI, nF>**

dove:

<N, AC> **è un grafo diretto con archi etichettati,**

$\{nI, nF\} \subseteq N$, $N - \{nI, nF\} = Ns \cup Np$

Ns e Np **sono insiemi disgiunti di *nodi istruzione* e *nodi predicato*;**

$AC \subseteq N-\{nF\} \times N-\{nI\} \times \{vero, falso, incond\}$ **rappresenta la relazione flusso di controllo;**

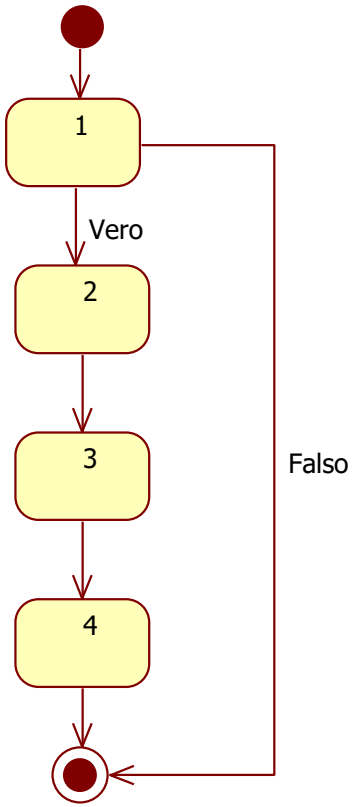
nI ed nF **sono detti rispettivamente *nodo iniziale* e *nodo finale*.**

•Un nodo $n \in Ns \cup \{nI\}$ **ha un solo successore immediato e il suo arco uscente è etichettato con incond.**

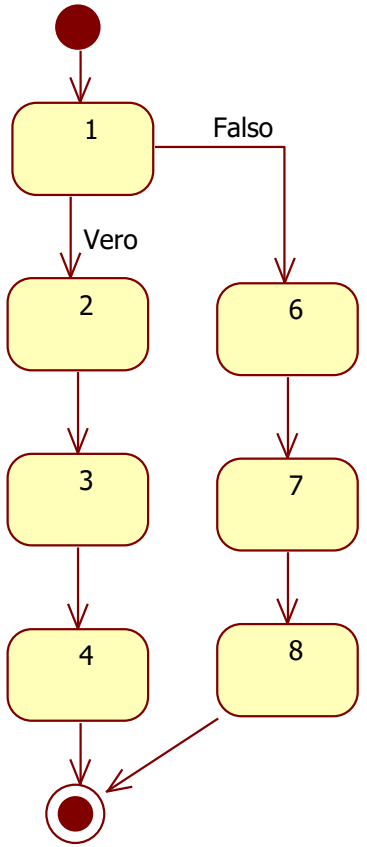
•Un nodo $n \in Np$ **ha due successori immediati e i suoi archi uscenti sono etichettati rispettivamente con vero e falso.**

Strutture di controllo e CFG

```
1.   if
     (decisione)
2.   {
3.     Blocco
4.   }
```



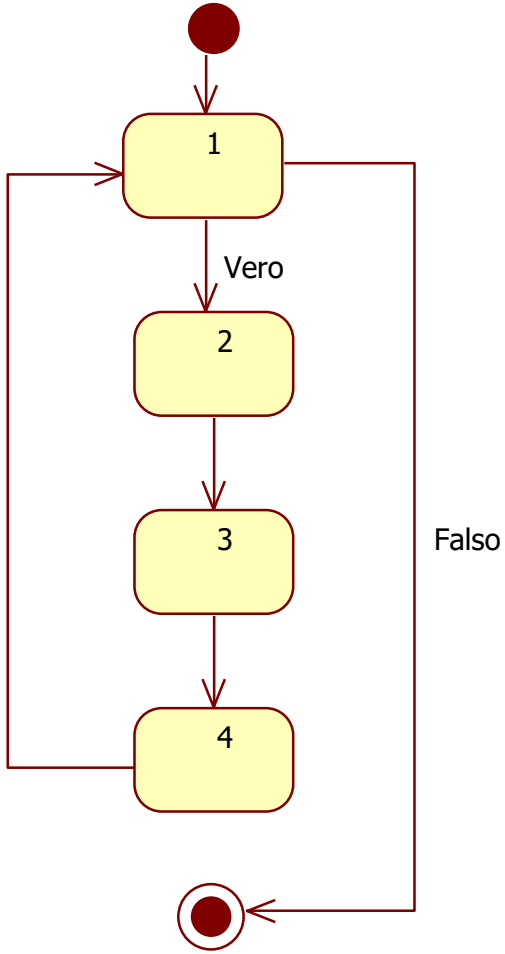

```
1.   if
     (decisione
     )
2.   {
3.     Blocco1
4.   }
5.   else
6.   {
7.     Blocco2
8.   }
```



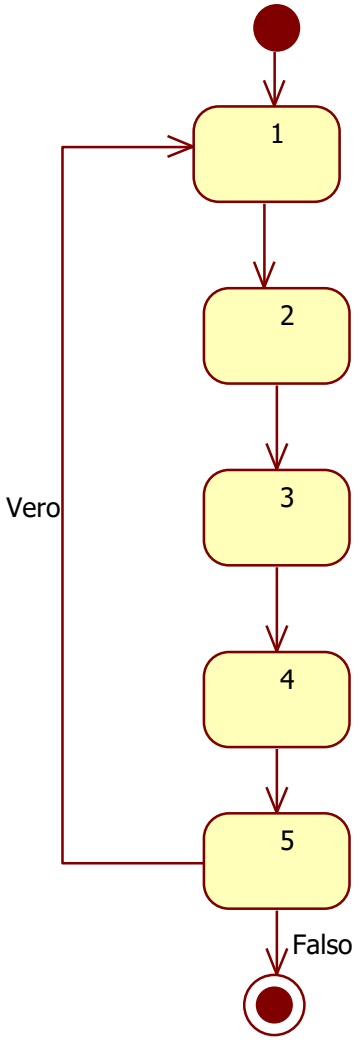
I nodi 2, 4, 6, 8 potevano anche essere omessi, dato che ad essi non corrisponde alcuna istruzione esecutiva, nel codice eseguibile del programma.
Uno switch può essere risolto trasformandolo (come fa il compilatore) in una serie di if else in cascata.

Strutture di controllo e CFG

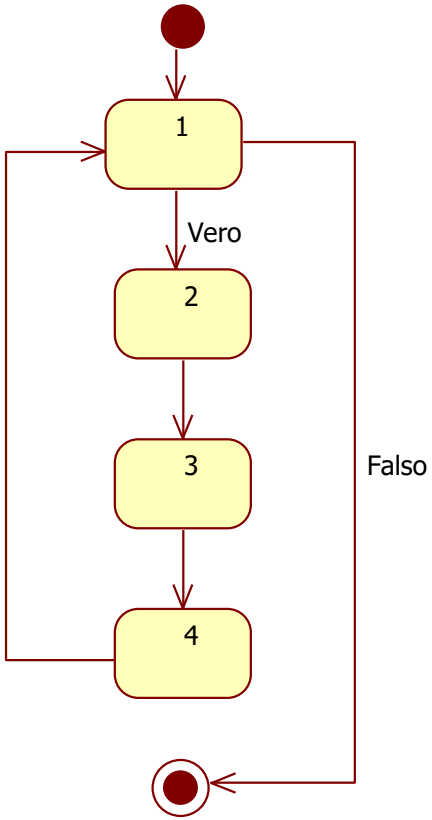
```
1.  while
    (decisione)
2.  {
3.    Blocco
4.  }
```



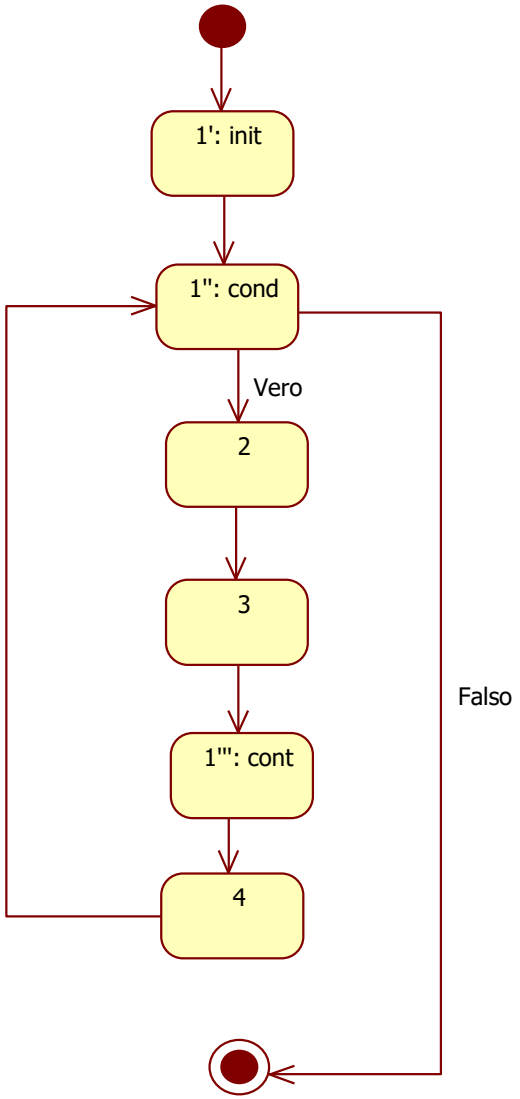

```
1.  do
2.  {
3.    Blocco
4.  }
5.  while
    (decisione
    )
```



Ciclo For e CFG



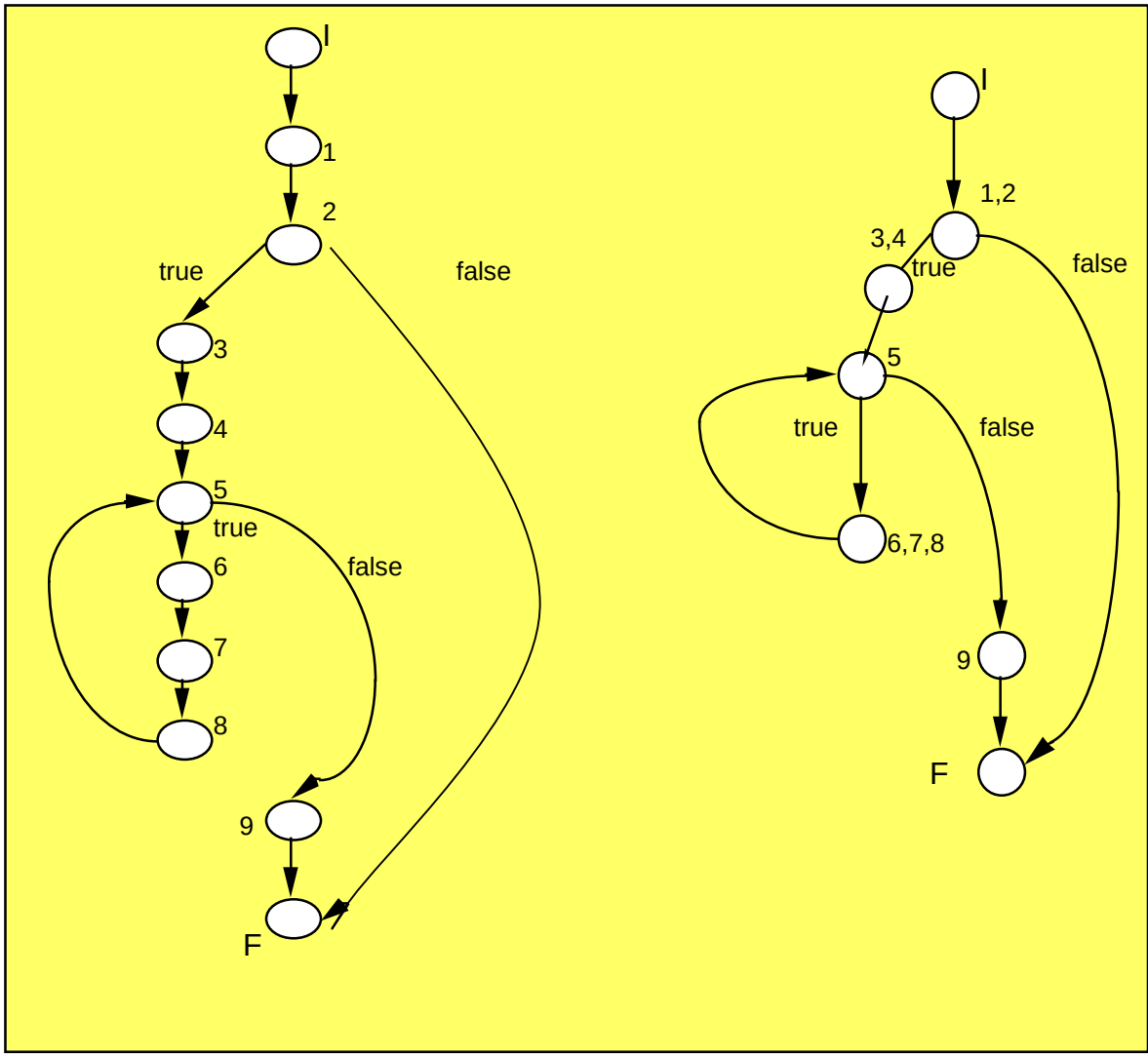

```
1.   for (init;decisione;cont)
2.   {
3.     Blocco
4.   }
```



Il CFG a destra esprime più precisamente la semantica del ciclo for, così come viene trasformato in codice oggetto da un compilatore C. Infatti ad ogni ciclo for corrisponde un init preventivo, una decisione all'inizio di ogni ciclo (come in un while) e un codice di continuazione cont (ad esempio i++) da ripetere prima di ricominciare il ciclo

Un esempio

```
procedure Quadrato;
  var x, y, n: integer;
  begin
1.   read(x);
2.   if x > 0
     then begin
3.          n := 1;
4.          y := 1;
5.          while x > 1 do
             begin
6.             n := n + 2;
7.             y := y + n;
8.             x := x - 1;
             end;
9.          write(y);
         end;
  end;
```



Criteri di copertura

- Copertura dei comandi (statement test)
  - Richiede che ogni nodo del CFG venga eseguito almeno una volta durante il testing;
  - è un criterio di copertura debole, che non assicura la copertura sia del ramo true che false di una decisione.

Criteri di copertura

- Copertura delle decisioni (branch test)
  - Richiede che ciascun arco del CFG sia attraversato almeno una volta;
    - *In questo caso ogni decisione è stata sia vera che falsa in almeno un test case*
  - un limite è legato alle decisioni in cui più condizioni (legate da operatori logici AND ed OR) sono valutate

Copertura delle condizioni (condition test)

- Ciascuna condizione nei nodi decisione di un CFG deve essere valutata sia per valori true che false.
- Esempio:

```
int check (x);// controlla se un intero è fra 0 e 100
int x;
{    if ((x>=0) && (x<= 200))
check= true;
     else check = false;
     }
```

TS={x=5, x=-5 } valuta la decisione sia per valori True che False, ma non le condizioni

TS1={x= 3, x=-1, x=210} è una Test suite che copre tutte le condizioni (ma non in tutte le combinazioni possibili poiché la combinazione (false,false) non può essere provata)

TS2={x=-4;x=300} è una Test Suite che copre tutte le condizioni ma non tutte le decisioni (l'if è sempre falso)

Copertura delle condizioni e decisioni

- Occorre combinare la copertura delle condizioni in modo da coprire anche tutte le decisioni.

- Es. if (x>0 && y>0) …
  - TS1={(x=2, y=-1), (x=-1, y=5)} copre le condizioni ma non le decisioni!
  - TS2={(x=2, y=1), (x=-1, y=-55)} copre sia le condizioni che le decisioni!
  - TS3={(x=2, y=1), (x=2, y=-1), (x=-2, y=1)(x=-1, y=-55)} copre tutte le combinazioni di tutte le condizioni (quindi anche tutte le condizioni e tutte le decisioni

Nota a margine

- Gli esempi precedenti riguardavano codice sorgente di linguaggi di alto livello
- In realtà, il CFG reale e le relative metriche di copertura dovrebbero essere valutate, per massimizzare la precisione, sul codice macchina generato dal codice sorgente
  - Fa eccezione il caso in cui vogliamo effettuare analisi statica, senza testing, direttamente di algoritmi, anziché di programmi
  - Per conoscere l'esatto codice macchina dobbiamo, però, conoscere le caratteristiche della macchina target e del compilatore
    - *Ad esempio, l'adozione di tecniche di ottimizzazione dei compilatori possono portare a variazioni dei CFG*
- In linguaggi come Java, è anche possibile una soluzione intermedia, con la copertura a livello di bytecode
- In conclusione, la misura di metriche di copertura sul codice di alto livello rappresenta una approssimazione, non sempre affidabile, delle analoghe misure sul codice eseguibile

Cammini linearmente indipendenti (McCabe)

- Un cammino è un'esecuzione del modulo dal nodo iniziale del CFG al nodo finale
- Un cammino si dice indipendente (rispetto ad un insieme di cammini) se introduce almeno un nuovo insieme di istruzioni o una nuova condizione
  - in un CFG un cammino è indipendente se attraversa almeno un arco non ancora percorso
- L'insieme di tutti i cammini linearmente indipendenti di un programma forma i cammini di base; tutti gli altri cammini sono generati da una combinazione lineare di quelli di base.
- Dato un programma, l'insieme dei cammini di base non è unico.

Numero di cammini linearmente indipendenti

- Il numero dei cammini linearmente indipendenti di un programma è pari al numero ciclomatico di McCabe:
  - V(G) = E - N +2
    - *Dove E: n.ro di archi in G - N: n.ro di nodi in G*
  - V(G) = P + 1
    - *Dove P: n.ro di predicati in G*
  - V(G) = n.ro di regioni chiuse in G + 1

- Test case esercitanti i cammini di base garantiscono l'esecuzione di ciascuna istruzione almeno una volta
- La copertura dei cammini linearmente indipendenti garantisce la copertura di tutti i cammini se non consideriamo il numero di volte in cui ogni ciclo può essere eseguito

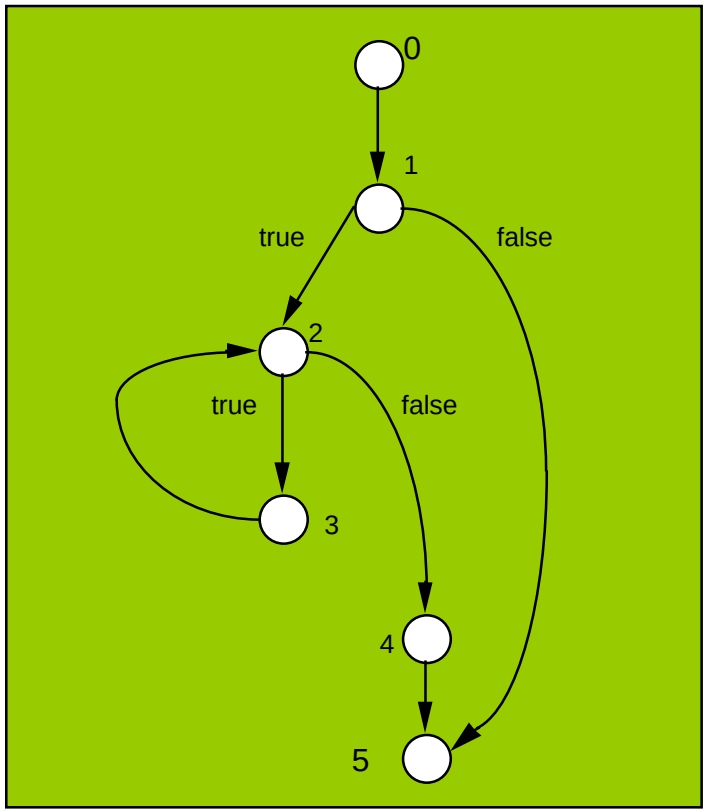
Thomas J. McCabe was born in Central Falls, RI, on November 28, 1941. He received the A.B. degree in mathematics from Providence College, Providence, RI and the M.S. degree in mathematics from the University of Connecticut, Storrs, in 1964 and 1966, respectively.

He has been employed since 1966 by the Department of Defense, National Security Agency, Ft. Meade, MD in various systems programming and programming management positions. He also, during a military leave, served as a Captain in the Army Security Agency engaged in large-scale compiler implementation and optimization. He has recently been active in software engineering and has developed and taught various software related courses for the Institute for Advanced Technology, the University of California, and Massachusetts State College System.

Mr. McCabe is a member of the American Mathematical Association.

http://www.mccabe.com/

Esempio



Complessità ciclomatica del programma è 3

V(G)= 3 =>3 cammini indipendenti

c1= 0-1-2-4-5
c2= 0-1-**2-3-2**-4-5
c3= 0-**1-5**

Ogni nuovo cammino è ottenuto negando una decisione di un cammino precedente e coprendo, quindi, almeno un nuovo arco

Criteri di copertura dei cammini

- Copertura dei cammini (path test)
  - spesso gli errori si verificano eseguendo cammini che includono particolari sequenze di nodi decisione
  - non tutti i cammini eseguibili in un CFG possono essere eseguiti durante il test (un CFG con loop può avere infiniti cammini eseguibili)
- Copertura dei cammini indipendenti
  - ci si limita ad eseguire un *insieme di cammini indipendenti* di un CFG, ossia un insieme di cammini in cui nessun cammino è completamente contenuto in un altro dell'insieme, nè è la combinazione di altri cammini dell'insieme
  - ciascun cammino dell'insieme presenterà almeno un arco non presente in qualche altro cammino
  - il numero di cammini indipendenti coincide con la complessità ciclomatica del programma

Relazioni tra i criteri di copertura

- La copertura delle decisioni implica la copertura dei nodi
- La copertura delle condizioni *non sempre* implica la copertura delle decisioni
- La copertura dei cammini linearmente indipendenti implica la copertura dei nodi e la copertura delle decisioni
- La copertura dei cammini è un test ideale ed implica tutti gli altri

Problemi

- E' possibile riconoscere automaticamente quale cammino linearmente indipendente viene coperto dall'esecuzione di un dato test case

- E' indecidibile il problema di trovare un test case che va a coprire un dato cammino
  - Alcuni cammini possono risultare non percorribili (*infeasible),* ma non è, in generale, possibile sapere se un cammino è percorribile

- La copertura dei cammini linearmente indipendenti non garantisce da errori dovuti, ad esempio, al numero di cicli eseguiti, per i quali sarebbe necessaria la copertura di tutti i cammini, che però rappresenta il testing esaustivo!
  - La copertura dei cammini linearmente indipendenti coincide con la copertura dei cammini in un programma senza cicli o in un programma in cui ogni ciclo è percorso sempre lo stesso numero di volte

EclEmma

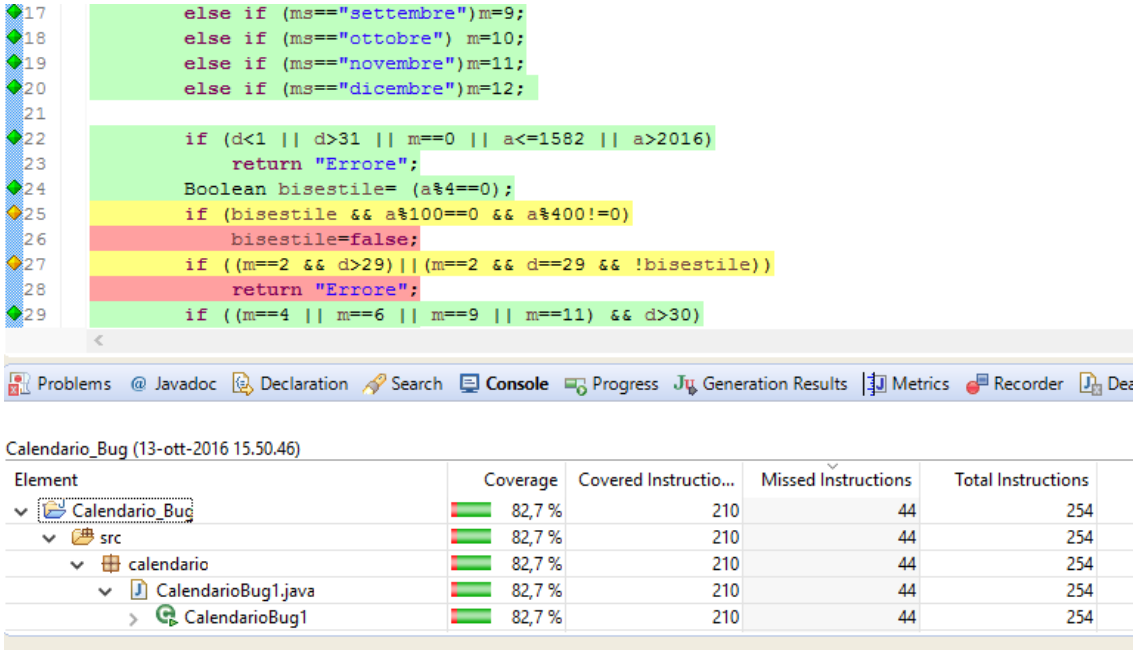
- Una estensione Eclipse per valutare la copertura del codice
  - http://eclemma.org
    - *Per eclipse:* **http://update.eclemma.org/**
    - *Attualmente integrata in alcune versioni (ad esempio Eclipse Oxygen)*
  - Istrumenta l'applicazione solo al tempo della valutazione della copertura
  - E' in grado di calcolare la copertura complessiva di più sessioni
  - Si basa su Emma, tool open source per la misura della copertura
    - *Si attiva aprendo l'opzione Coverage as nel menu contestuale dell'applicazione da testare*
    - *I risultati si possono leggere dal menu contestuale (Properties/Coverage) o da apposita finestra*

EclEmma

- Emma registra tutte le esecuzioni di ogni istruzione del bytecode
- Associa le istruzioni di bytecode alle corrispondenti istruzioni nel codice sorgente
- Genera report della copertura delle righe di codice sorgente
  - Il valore di copertura può essere:
    - *1: tutte le istruzioni del corrispondente bytecode sono state coperte*
    - *Frazionario, compreso tra 0 e 1: solo una parte delle istruzioni sono state coperte*
    - *0: nessuna istruzione del bytecode è stata coperta*

EclEmma

- In particolare, può essere generato un report che «colora» le righe del codice sorgente:
  - In verde: tutto il bytecode coperto
  - In giallo: copertura parziale
  - In rosso: non coperto



Altri strumenti per la misura della copertura

- Emma (http://emma.sourceforge.net/) è lo strumento su cui si basa EclEmma. E' eseguibile da linea di comando, funziona a livello di JVM senza istrumentazione del codice sorgente. Tra le funzionalità messe a disposizione a linea di comando, la possibilità di fondere la copertura ottenuta con due sessioni di esecuzione e la generazione di un report html con tutto il codice coperto

- Codecover (http://codecover.org/index.html), vedi Appendice
  - E' in grado di fornire informazioni più dettagliate (ad esempio il numero di volte in cui una riga è stata eseguita, il numero di volte che un ciclo è stato eseguito ed altro)

- Cobertura (http://cobertura.github.io/cobertura/)

- Altri strumenti (per Java): http://en.wikipedia.org/wiki/Java_Code_Coverage_Tools

Nota: spesso i tool di copertura sono incompatibili tra loro, poiché introducono molteplici istrumentazioni

Traccia di tesina di approfondimento teorico/pratico

- Una funzionalità quasi mai messa a disposizione dagli strumenti di misura della copertura è il conteggio di quante volte sia stata coperta una riga di codice e da quali test.
  - Tale misura può essere molto utile per valutare la difficoltà di copertura delle righe di codice e la ridondanza dei test
  - L'unico strumento che fornisce tale dato è codecover, ma non è aggiornato alle ultime versioni di Java
- Cercare o realizzare strumenti per la misura della copertura che forniscano anche report sul numero di volte in cui ogni riga è stata coperta
  - La realizzazione può basarsi anche sul riutilizzo di strumenti esistente come emma e sull'analisi dettagliata dei loro risultati
  - Un primo strumento valutabile è Atlassian Clover

Esercizio

- Per la classe calendar, visualizzare il livello di copertura ottenuto con i test JUnit progettati in precedenza
- Scrivere ulteriori casi di test in grado di coprire totalmente, secondo i diversi criteri di copertura studiati, gli elementi del metodo calend
- Sfruttare, eventualmente, le informazioni di copertura per trovare i difetti relativi agli eventuali malfunzionamenti trovati

Testing White box e valori di test

- Per sfruttare al meglio le potenzialità del testing white box dovremmo pensare di creare test a partire dal codice.
- Ad esempio, leggendo il codice del calendario, i seguenti valori di test sembrerebbero da provare:
  - Giorno: <1, >31, ==29, >29, ==30
  - Mese: Gennaio, Febbraio, marzo, aprile, maggio, giugno, luglio, agosto, settembre, ottobre, novembre, dicembre
  - Anno: <=1582, >2016, %100==0, %400==0

```
public static String calend(int d, String ms, int a)
{
int m=0;
if (ms=="Gennaio") m=1;
else if (ms=="Febbraio") m=2;
else if (ms=="marzo") m=3;
else if (ms=="aprile")m=4;
else if (ms=="maggio")m=5;
else if (ms=="giugno")m=6;
else if (ms=="luglio")m=7;
else if (ms=="agosto")m=8;
else if (ms=="settembre")m=9;
else if (ms=="ottobre") m=10;
else if (ms=="novembre")m=11;
else if (ms=="dicembre")m=12;

if (d<1 || d>31 || m==0 || a<=1582 || a>2016)
return "Errore";
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 "Errore";
if ((m==4 || m==6 || m==9 || m==11) && d>30)
return "Errore";


if (m<=2)
 {
  m = m + 12;
  a--;
 };
 int f1 = a / 4;
 int f2 = a / 100;
 int f3 = a / 400;
 int f4 = (int) (2 * m + (.6 * (m + 1)));
 int f5 = a + d + 1;
 int x = f1 - f2 + f3 + f4 + f5;
 int k = x / 7;
 int n = x - k * 7;

 if (n==1) return "Lunedi";
 else if (n==2) return "Martedi";
 else if (n==3) return "Mercoledi";
 else if (n==4) return "Giovedi";
 else if (n==5) return "Venerdi";
 else if (n==6) return "Sabato";
 else if (n==0) return "Domenica";
 else return "Errore";
}
```

Considerazioni

- Utilizzando casi di test ottenuti dal codice:
  - Si scoprono combinazioni nuove, quali l'importanza degli anni il cui valore è divisibile per 400 e per 4
  - Si scoprono combinazioni inutili quali Gennaio e Febbraio scritti erroneamente con le maiuscole (la mancata copertura del codice avrebbe già dovuto condurci a correggere i due bug)
- Non troviamo invece i valori gennaio e febbraio (a causa di due bug), per cui li proveremmo solo con approccio black box
- Purtroppo non scopriamo l'importanza del mese di febbraio, poiché i controlli sono fatti sulla variabile intera m e non sull'input stringa ms … dobbiamo trovare il modo di metterli in relazione

Traccia di tesina di approfondimento teorico/pratico

- Strumenti di analisi statica del codice possono aiutarci a progettare casi di test white box
- Uno strumento molto potente ed estendibile che supporta l'analisi del codice con finalità di test white box è Java Path Finder
  - http://babelfish.arc.nasa.gov/trac/jpf
- Una possibile tesina consiste nello studiare Java Path Finder, valutarne estensioni esistenti in grado di supportare il testing white box ed eventualmente svilupparne altre

Appendice

CodeCover

- CodeCover è un plug-in open source per Eclipse realizzato dall'università di Stoccarda e reperibile a: http://codecover.org/index.html
- **Consente di valutare metriche di copertura**
  - CodeCover measures statement, branch, loop, term coverage (subsumes MC/DC), question mark operator coverage, and synchronized coverage.
- Scritto per Java e Cobol
- Si può utilizzare sia in modalità standalone che sotto Eclipse

CodeCover per Eclipse

- Può essere installato semplicemente aggiungendolo come plug-in aggiuntivo reperibile all'indirizzo:
  http://update.codecover.org/

- Un tutorial d'esempio è reperibile all'indirizzo:
  http://codecover.org/documentation/tutorials/how_to_complete.html

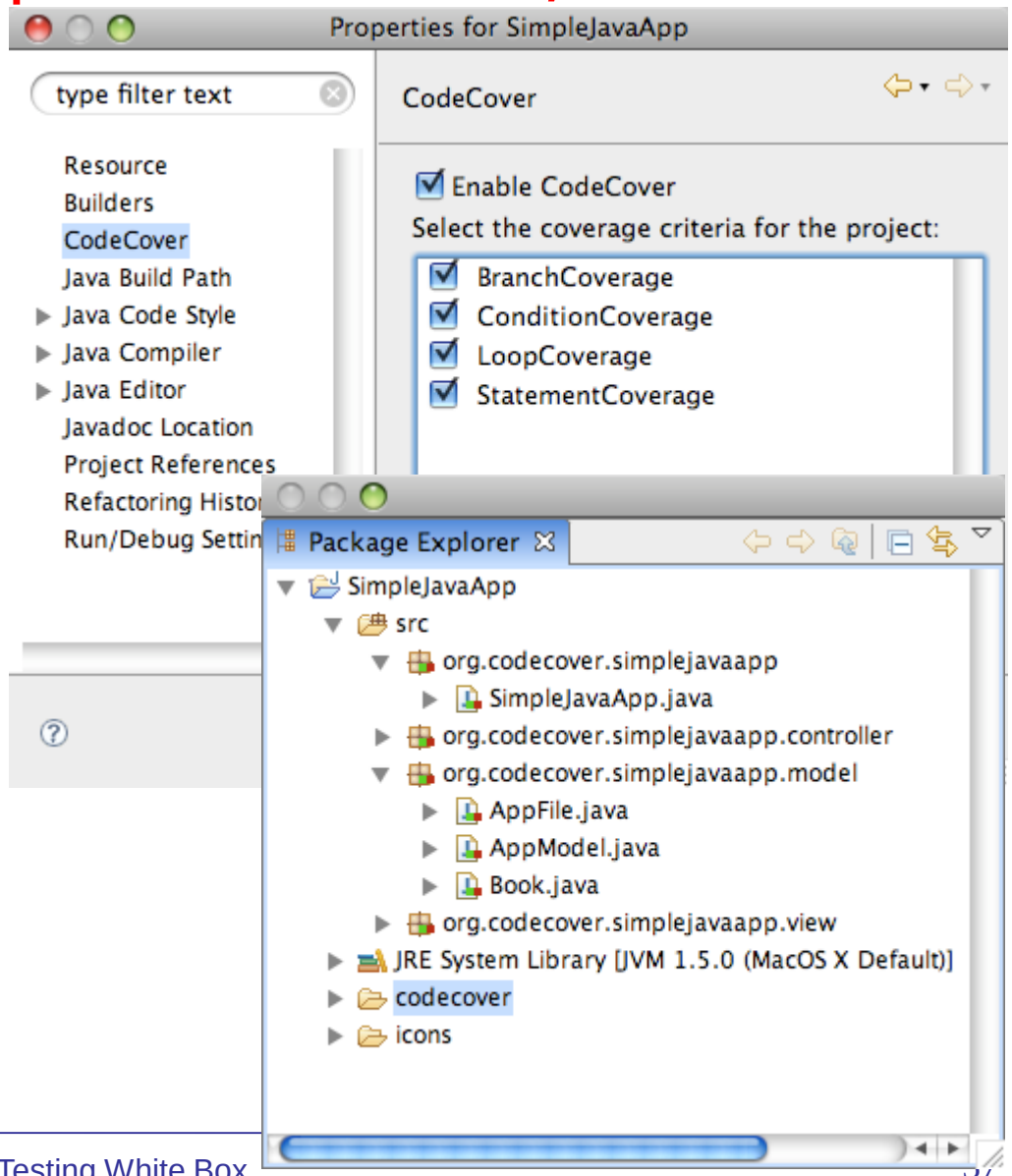
CodeCover Features

- Può misurare tanto la copertura per esecuzioni di un'applicazione per la quale sia stato attivato preventivamente il monitoraggio, sia per esecuzioni di test case scritti con Junit

- Mostra molte statistiche riguardanti l'esito dei casi di test e consente di generare report HTML con tali risultati

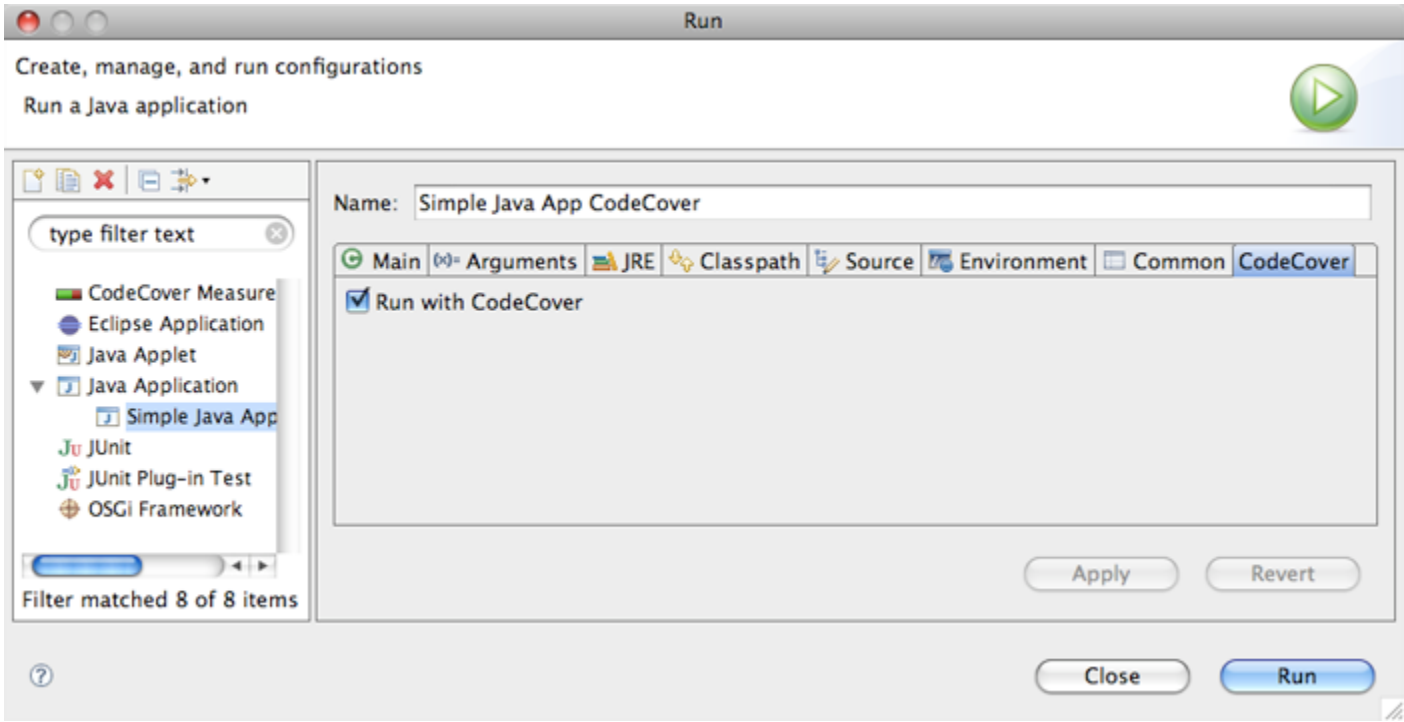
Mini Tutorial per CodeCover 1/5

- Abilitare CodeCover tra le proprietà del progetto
  - Causa l'instrumentazione del bytecode
- Nel menu contestuale relativamente ad ogni package/file, abilitare la misura



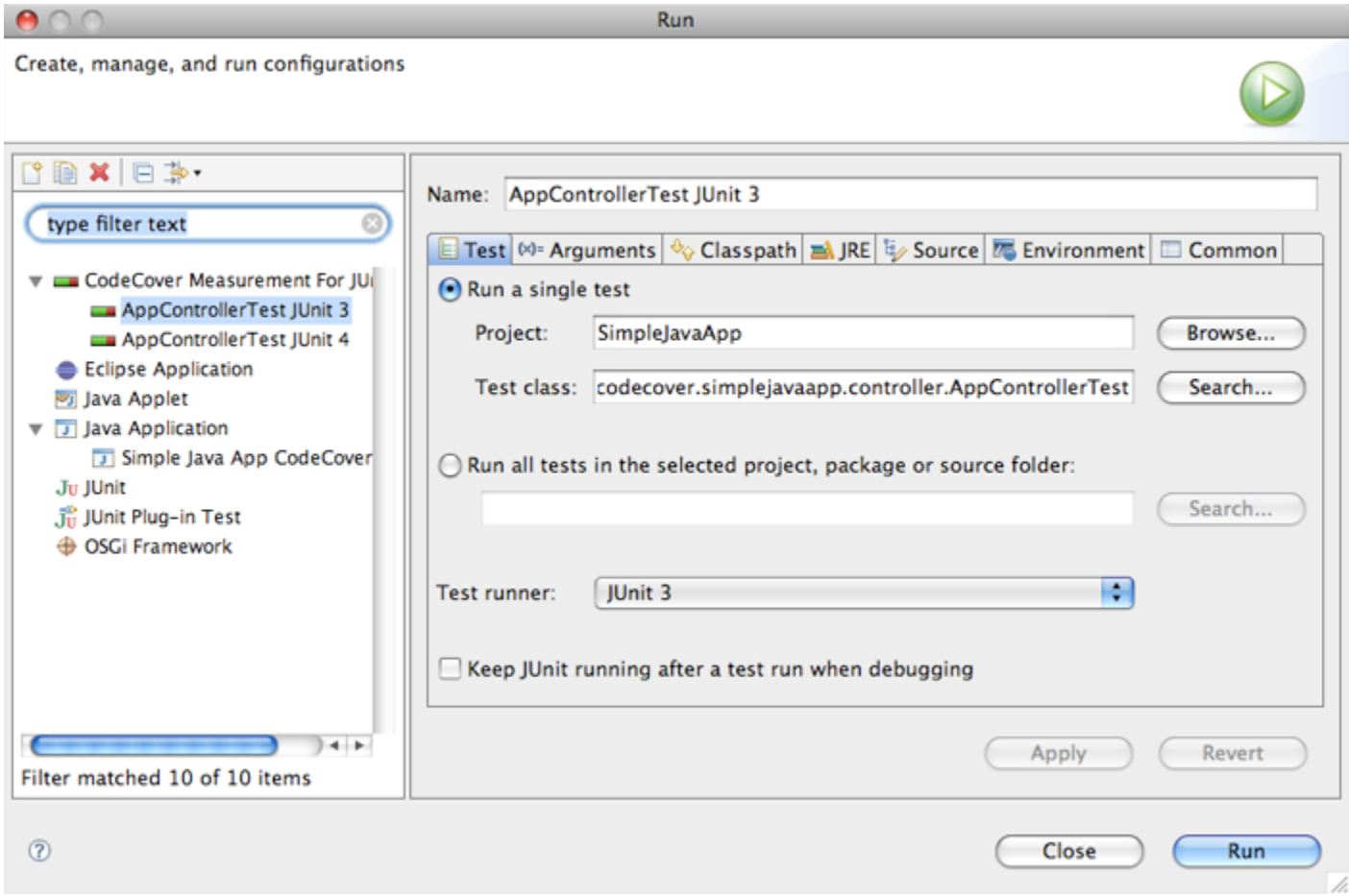
Mini Tutorial per CodeCover 2/5

- Nelle proprietà del profilo di esecuzione abilitare CodeCover
- Abilitare da Windows/Show View/Other le varie view di Codecover:
  - Test Sessions
  - Coverage
  - Correlation
  - …



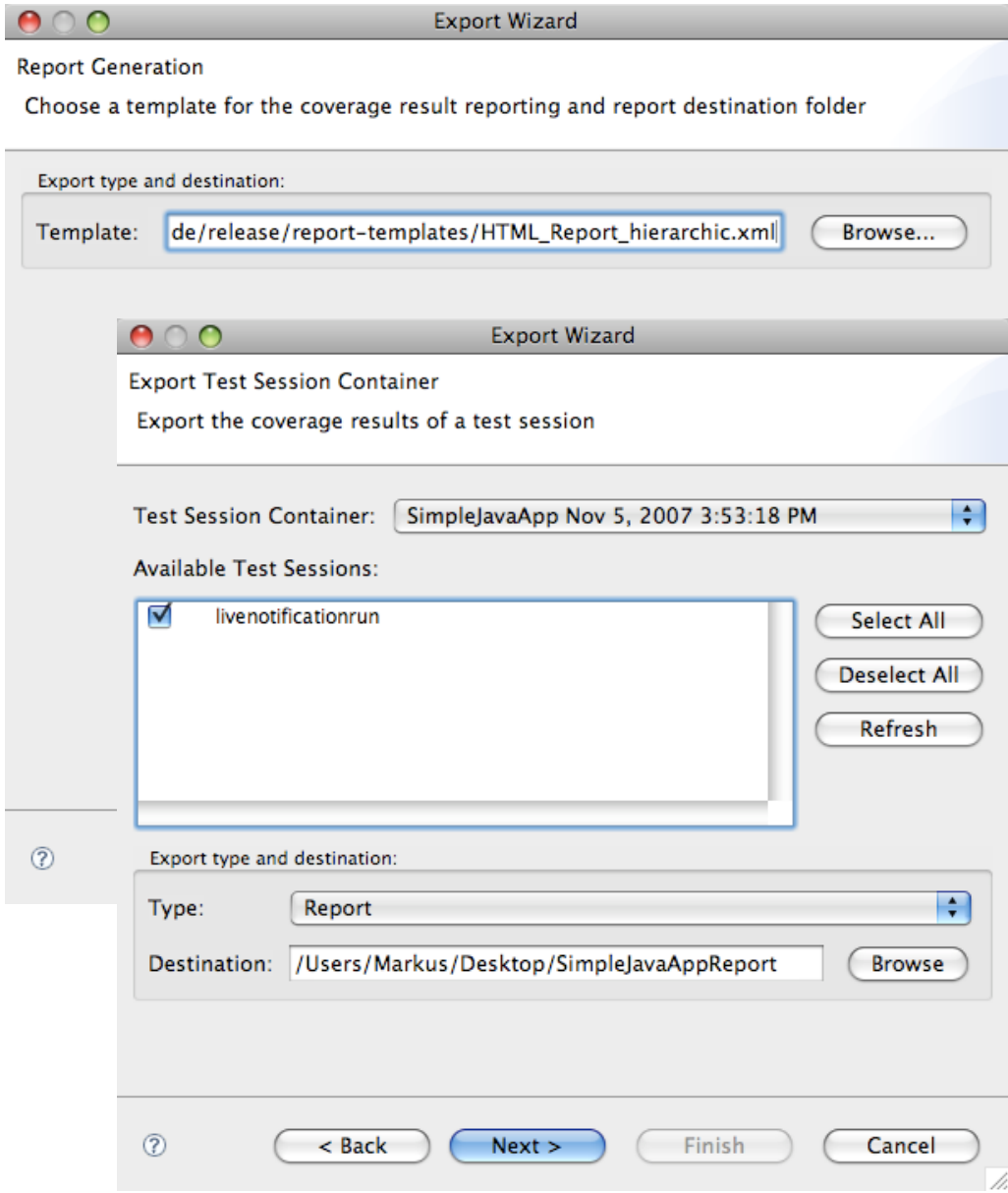
Mini Tutorial per CodeCover 3/5

- Per conoscere la copertura raggiunta con test case Junit è sufficiente creare un profilo di esecuzione dei test sotto CodeCover Measurement for JUnit



Mini Tutorial per CodeCover 4/5

- Per esportare i risultati in HTML è sufficiente utilizzare il wizard di esportazione in File/Export



Mini Tutorial per CodeCover 5/5

- La copertura delle righe del codice può essere vista anche direttamente sul codice
- Rosso: non coperta
- Giallo: coperta parzialmente (decisione)
- Verde: coperta

AppController.java

```
private final class AppViewListener implements ViewListener {

    /**
     * (non-Javadoc)
     *
     * @see org.codecover.simplejavaapp.view.AppView.ViewListener#update(org.codecover.simplejavaapp.view.Ap
     *      java.lang.String, java.lang.Object)
     */
    public void update(ViewEvent viewEvent, String fileId, Object args) {
        switch (viewEvent) {
            case EVENT_CLOSE_WINDOW:
                onCloseWindow(fileId);
                break;
            case EVENT_DELETE:
                onDelete(fileId, (String[]) args);
                break;
            case EVENT_NEW_BOOK:
                onNewBook(fileId);
                break;
            case EVENT_NEW_WINDOW:
                onOpenNewWindow();
                break;
            case EVENT_OPEN:
                onOpen(fileId);
                break;
            case EVENT_QUIT:
                onQuit(fileId, (String[]) args);
                break;
            case EVENT_SAVE:
                onSave(fileId);
                break;
            case EVENT_SAVE_AS:
                onSaveAs(fileId);
                break;
        }
    }
}
```

CodePro Analytix

Google CodePro AnalytiX™

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

Features

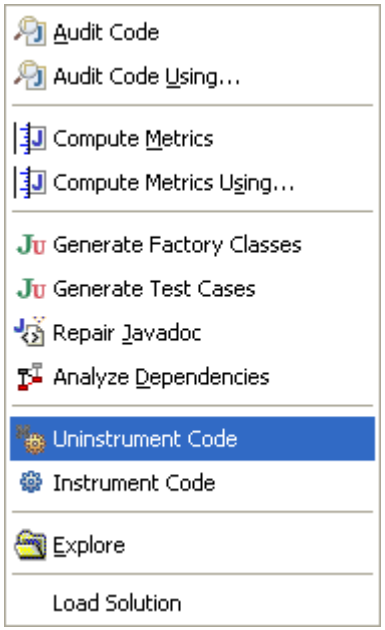
- CodePro Analytix offre features relative a:
  - Analisi del codice
  - Generazione automatica di casi di test JUnit
  - Metriche di qualità del prodotto
  - Misura della code coverage
  - Costruzione del grafo delle dipendenze
  - …

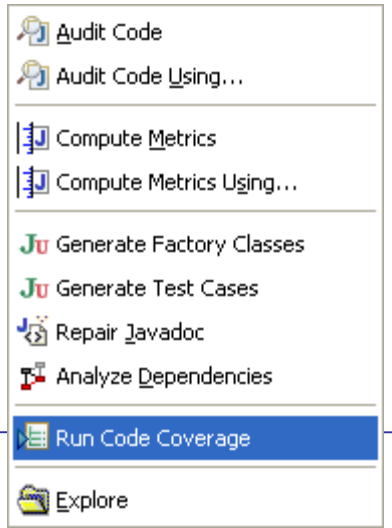
Code Coverage con CodePro

- CodePro fornisce funzionalità per la misura automatica della copertura di codice in termine di righe, blocchi, metodi, classi ottenuta a seguito dell'esecuzione del main oppure di casi di test Junit
- CodePro si basa su Emma (così come EclEmma)
- E' in grado di generare report dettagliati della copertura ottenuta

Utilizzo di Code Coverage CodePro

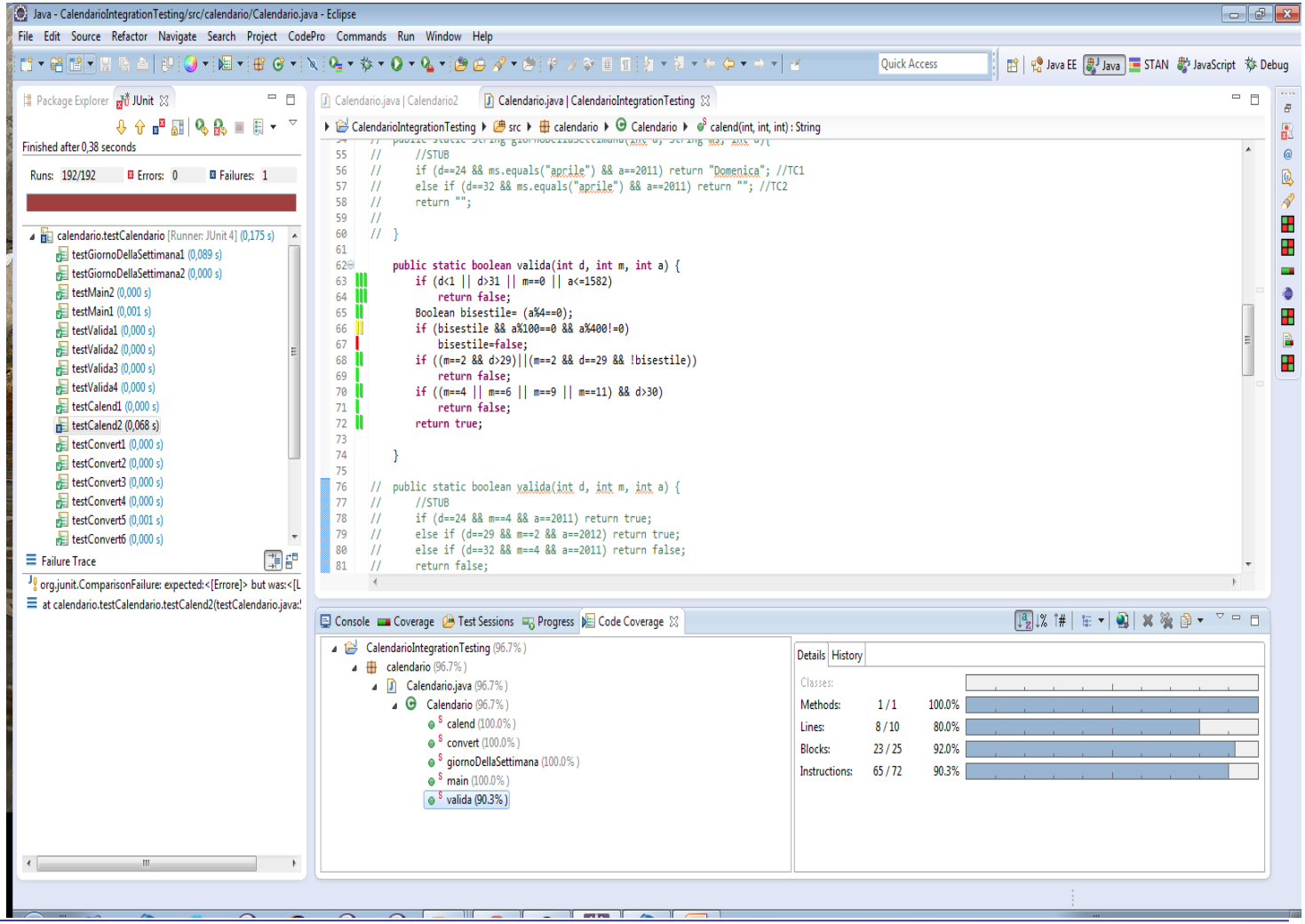
1. Dato un progetto, instrumentare il codice a supporto della misura, scegliendo l'apposita opzione (*Instrument Code*) nel menu contestuale CodePro Tools
   - L'opzione Uninstrument consente di togliere l'instrumentazione
2. Eseguire il progetto normalmente (anche tramite casi di test)
   - In alcuni casi è necessario eseguire il progetto tramite l'opzione *Run Code Coverage*
3. Aprire la View denominata CodeCoverage





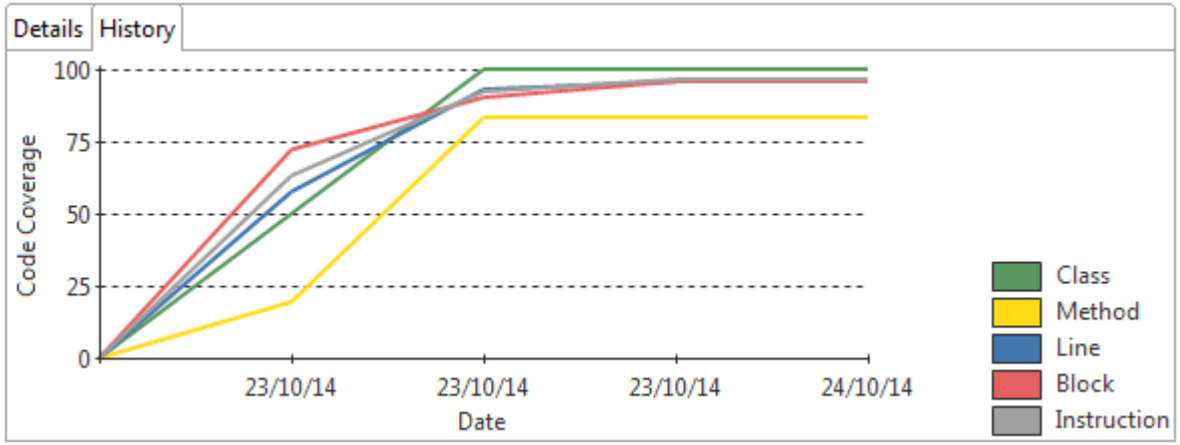
Utilizzo di Code Coverage CodePro

- Vengono mostrate le righe coperte (verdi), non coperte (rosse) , parzialmente coperte (gialle).
- Nel caso di righe corrispondenti a più di una riga del bytecode, un numero di | pari alle avvenute coperture è visualizzato.



Utilizzo di Code Coverage CodePro

- Nella scheda history si può vedere l'andamento della copertura a seguito delle ultime test suite eseguite



- E' possibile salvare un report HTML della copertura

```
61
62      public static boolean valida(int d, int m, int a) {
63          if (d<1 || d>31 || m==0 || a<=1582)
64              return false;
65          Boolean bisestile= (a%4==0);
66          if (bisestile && a%100==0 && a%400!=0)
67              bisestile=false;
68          if ((m==2 && d>29)||(m==2 && d==29 && !bisestile))
69              return false;
70          if ((m==4 || m==6 || m==9 || m==11) && d>30)
71              return false;
72          return true;
73
```

Avvertenze

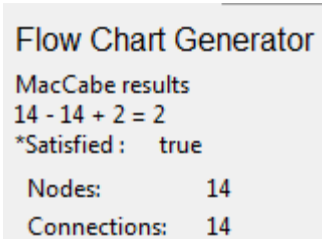
- L'instrumentazione di Code Coverage per CodePro Analytix è di solito incompatibile con quella di CodeCoverage
- L'instrumentazione è incompatibile con la funzionalità di Test Case Generation di CodePro
  - Per generare test su di un codice instrumentato, va prima deinstrumentato, poi si possono generare i test, infine si può instrumentarlo nuovamente ed eseguire i test appena generati

Eclipse Control Flow Graph Generator

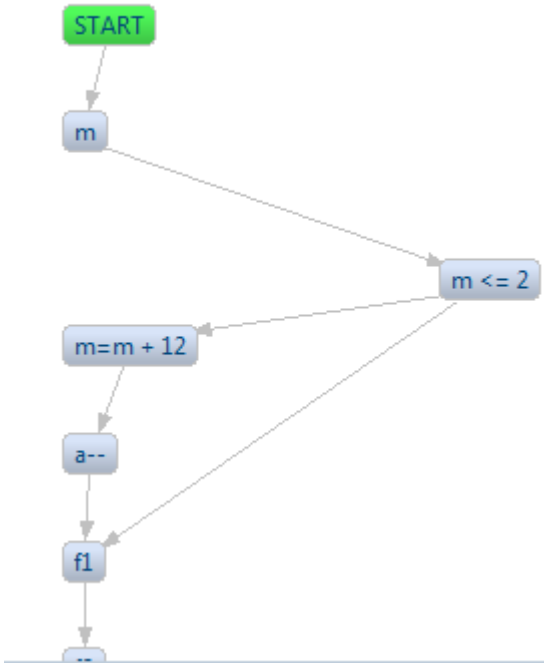
- http://eclipsefcg.sourceforge.net/
- E' un'estensione open source di Eclipse che consente di disegnare CFG di metodi di applicazioni Java
- Può essere installato semplicemente aggiungendolo come plug-in aggiuntivo reperibile all'indirizzo: **http://eclipsefcg.sourceforge.net/**
- Per utilizzarlo basta selezionare l'opzione CFG Generator dal menu contestuale di un qualsiasi metodo
- Si integra anche con CodeCover

Mini Tutorial per CFG Generator

- E’ sufficiente scegliere CFG Generator/ Build nel menu contestuale di un metodo



- In alternativa, è anche possibile generare il CFG da una sessione di test la cui copertura è stata salvata con CodeCover

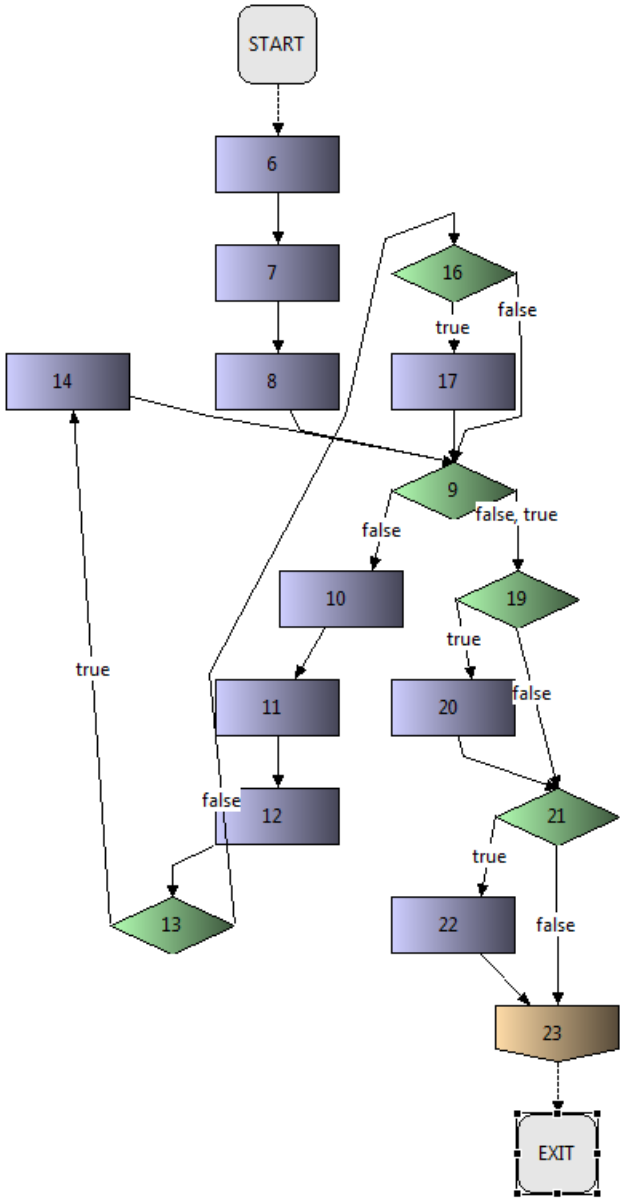


Dr. Garbage

- Dr. Garbage (http://www.drgarbage.com/index.html) è un'altra suite di strumenti open source per eclipse che sono in grado di ricostruire il Control Flow Graph
  - **Per installarlo sotto Eclipse (Mars)**
    https://sourceforge.net/projects/drgarbagetools/files/eclipse/4.5/stable
- Dr. Garbage consente di ricostruire il Control Flow Graph a tre livelli:
  - A livello di source code
  - A livello di bytecode
  - A livello di blocchi di bytecode
- Tutorial completi sono sul sito di Dr. Garbage
  - http://www.drgarbage.com/howto/cfgf-tutorial/

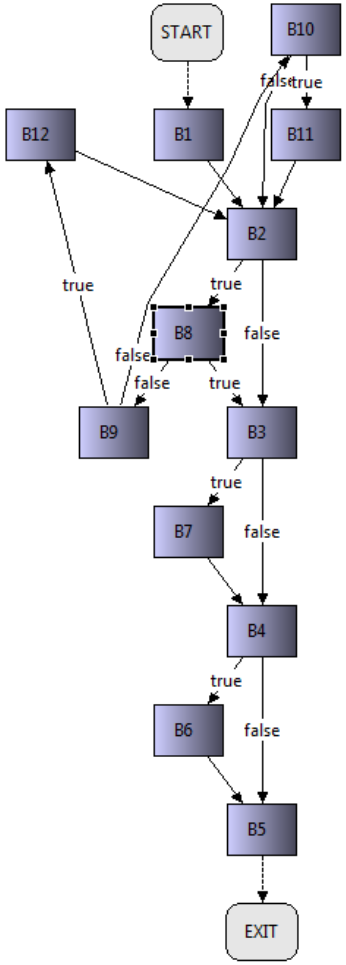
Dr. Garbage

```
6.  int tent=0;
7.  int x=7;
8.  int num=0;
9.      while ((tent<4)&&(num!=x)) {
10.       System.out.println("Indovina il numero :");
11.       num=System.in.read();
12.       tent++;
13.       if (num>x)
14.       System.out.println("Un po' di meno");
15.       else
16.           if (num<x)
17.           System.out.println("Un po' di piu'");
18.     }
19.     if (num==x)
20.     System.out.println("Bravo");
21.     if (tent==4)
22.     System.out.println("Hai perso!");
23.     return 0;
```



Dr. Garbage

- CFG a livello bytecode blocks



- CFG a livello bytecode