Testing & Verifica del SW

Esame del 20/06/2015 di Marziali Michel

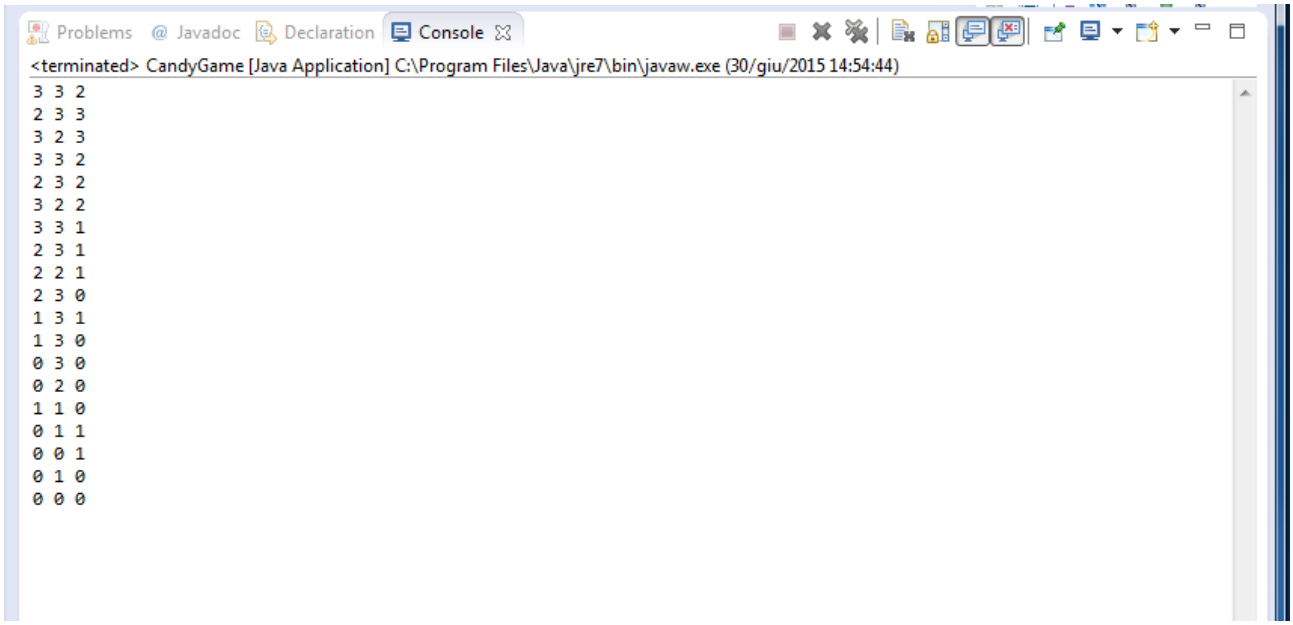
Esercizio1

L'esercizio numero 1 consisteva nella realizzazioni della classe *CandyGame* in grado di simulare il gioco descritto nel tema d'esame e testarla attraverso il tool JUNIT.

Il primo test effettuato si basa su di un semplice utilizzo del metodo main per esser sicuri che la funzione game() funzioni.

Il risultato ottenuto alla Console è il seguente:

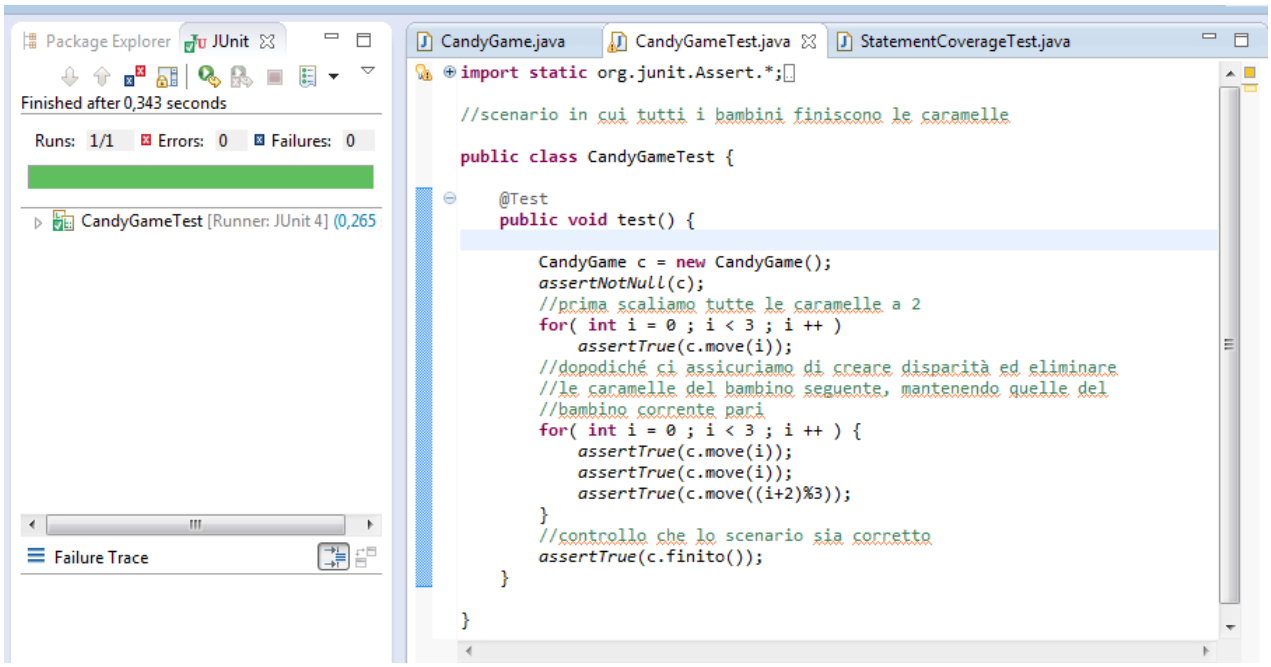
```
public void game() {
    Random r = new Random();
    while( !finito() ) {
        move(r.nextInt(3));
        System.out.println(toString());
    }
}

@Override
public String toString() {
    String s = "";
    for (int i = 0; i < bambini.length; i++) {
        s += bambini[i] +" ";
    }
    return s;
}

public static void main(String[] args) {
    CandyGame c = new CandyGame();
    c.game();
}
```



Il test seguente consiste, come da traccia, nella creazione della classe di test CandyGameTest col fine di assicurarsi che il sistema possa giungere a una conclusione.

Testing & Verifica del SW

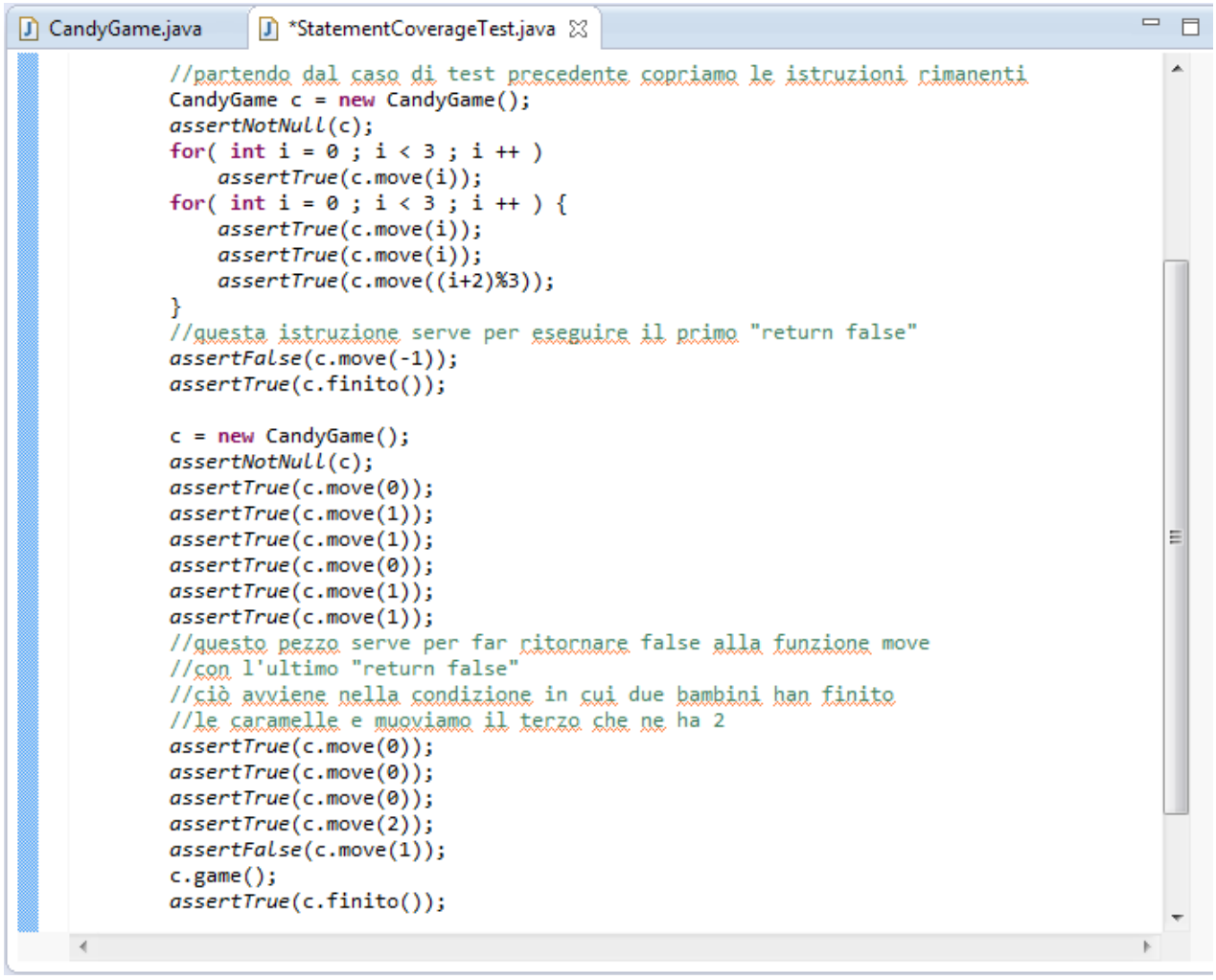
Esame del 20/06/2015 di Marziali Michel



Il risultato, come possiamo notare dalla barra verde, è positivo.

A questo punto analizziamo i test di copertura.

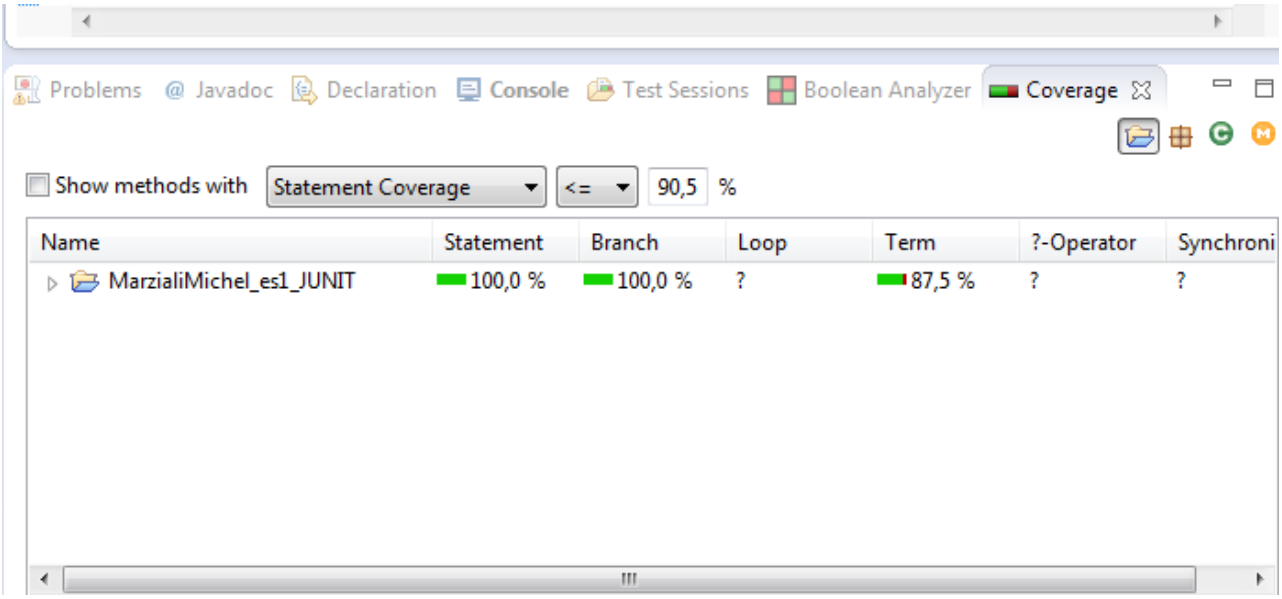
Lo Statement Coverage, il Branch Coverage ed il Decision Coverage in questo caso coincidono tutti e sono inseriti nella classe di test JUNIT StatementCoverageTest, il quale è stato creato a partire dalla classe di test precedente.



Testing & Verifica del SW

Esame del 20/06/2015 di Marziali Michel

Eseguendola con CodeCover otteniamo il seguente risultato:



CandyGame.java *StatementCoverageTest.java

```java
import java.util.Random;


public class CandyGame {

    //array contenente le caramelle possedute da ogni bambino
    private int[] bambini;

    //inizializzo la struttura del gioco
    public CandyGame() {
        bambini = new int[]{ 3 , 3 , 3 };

    }

    //metodo necessario ad effetturare il movimento
    //del bambino di posto child
    //N.B. vengono sfruttati i moduli per effettuare
    //i movimenti all'interno dell'array in modo sicuro
    public boolean move( int child ) {
        //prima di chiamare il metodo controllo che
        //il gioco non sia finito e l'input sia corretto
        if(finito() || child<0 || child >= 3)
            return false;
        //punto 1
        if( bambini[child] >= 1 && bambini[child] != 2 ) {
            bambini[child] --;
            return true;
        }
        //punto 2.1
        if(bambini[(child+2)%3]>0) {
            bambini[(child+2)%3]--;
            bambini[child]++;
            return true;
```

Testing & Verifica del SW

Esame del 20/06/2015 di Marziali Michel

CandyGame.java *StatementCoverageTest.java

```
        //punto 1
        if( bambini[child] >= 1 && bambini[child] != 2 ) {
            bambini[child] --;
            return true;
        }
        //punto 2.1
        if(bambini[(child+2)%3]>0) {
            bambini[(child+2)%3]--;
            bambini[child]++;
            return true;
        }
        //punto 2.2
        if(bambini[(child+1)%3]>0) {
            bambini[(child+1)%3]--;
            bambini[(child+2)%3]++;
            return true;
        }
        return false;
    }

    //metodo che segnala la fine del gioco
    public boolean finito() {
        return  bambini[0] == 0 &&
                bambini[1] == 0 &&
                bambini[2] == 0 ;
    }

    public void game() {
        Random r = new Random();
        while( !finito() ) {
            move(r.nextInt(3));
            //System.out.println(toString());
        }
    }
```

Il test successivo prevede la copertura MCDC della decisione

`(finito() || child<0 || child >= 3)`

presente all'inizio della funzione “move”.

Come si evince dal codice e dai commenti, vengono valutati il caso in cui la decisione sia falsa e tutti i casi in cui le singole condizioni rendono la decisione vera.

Testing & Verifica del SW

Esame del 20/06/2015 di Marziali Michel

CandyGame.java | MCDCTest.java

```
@Test
public void test() {
    CandyGame c = new CandyGame();
    assertNotNull(c);
    //test1
    //finito()=false    childe<0=false      child>=3=false
    //result=false(quindi l'istruzione dovrebbe tornare true continuando la sua esecuz
    assertTrue(c.move(2));//test1
    //test2
    //finito()=false    childe<0=true       child>=3=false
    //result=true(quindi l'istruzione dovrebbe tornare false)
    assertFalse(c.move(-2));//test1
    //test3
    //finito()=false    childe<0=false      child>=3=true
    //result=true(quindi l'istruzione dovrebbe tornare false)
    assertFalse(c.move(22));
    //test4
    //finito()=true     childe<0=false      child>=3=false
    //result=true(quindi l'istruzione dovrebbe tornare false)
    //devo far concludere il gioco perchè si verifichi,
    //dunque riutilizzo lo schema di CandyGameTest
    c = new CandyGame();
    assertNotNull(c);
    for( int i = 0 ; i < 3 ; i ++ )
        assertTrue(c.move(i));
    for( int i = 0 ; i < 3 ; i ++ ) {
        assertTrue(c.move(i));
        assertTrue(c.move(i));
        assertTrue(c.move((i+2)%3));
    }
    assertTrue(c.finito());
    //ecco il risultato sperato
    assertFalse(c.move(2));
}
```

Problems | Javadoc | Declaration | Console | Test Sessions | Boolean Analyzer | Coverage

Class: CandyGame Condition: ((finito() OR child<0) OR child >= 3)

(	(	finito()	OR	child<0	)	OR	child >= 3	)	Result	Test Cases (Number of Executions)
		F	T	T		T	x		1	MCDCTest:test (1) Coverage: 16,0
		F	F	F		F	F		0	MCDCTest:test (13) Coverage: 50,0
		T	T	x		T	x		1	MCDCTest:test (1) Coverage: 16,0
		F	F	F		T	T		1	MCDCTest:test (1) Coverage: 16,0

term coverage for all test cases: 100,0 %

Sarebbe stato necessario, infine, eseguire test successivi che garantissero la Condition Coverage al 100%, ma l'esecuzione tramite randoop ha fortunatamente dato come esito la copertura totale di Statement, Branch e Condition.

Testing & Verifica del SW

Esame del 20/06/2015 di Marziali Michel

CandyGame.java | MCDCTest.java | RandoopTest.java

```java
import java.util.Random;


public class CandyGame {

    //array contenente le caramelle possedute da ogni bambino
    private int[] bambini;

    //inizializzo la struttura del gioco
    public CandyGame() {
        bambini = new int[]{ 3 , 3 , 3 };

    }

    //metodo necessario ad effetturare il movimento
    //del bambino di posto child
    //N.B. vengono sfruttati i moduli per effettuare
    //i movimenti all'interno dell'array in modo sicuro
    public boolean move( int child ) {
        //prima di chiamare il metodo controllo che
        //il gioco non sia finito e l'input sia corretto
        if(finito() || child<0 || child >= 3)
            return false;
        //punto 1
        if( bambini[child] >= 1 && bambini[child] != 2 ) {
            bambini[child] --;
            return true;
        }
        //punto 2.1
        if(bambini[(child+2)%3]>0) {
            bambini[(child+2)%3]--;
            bambini[child]++;
            return true;
```

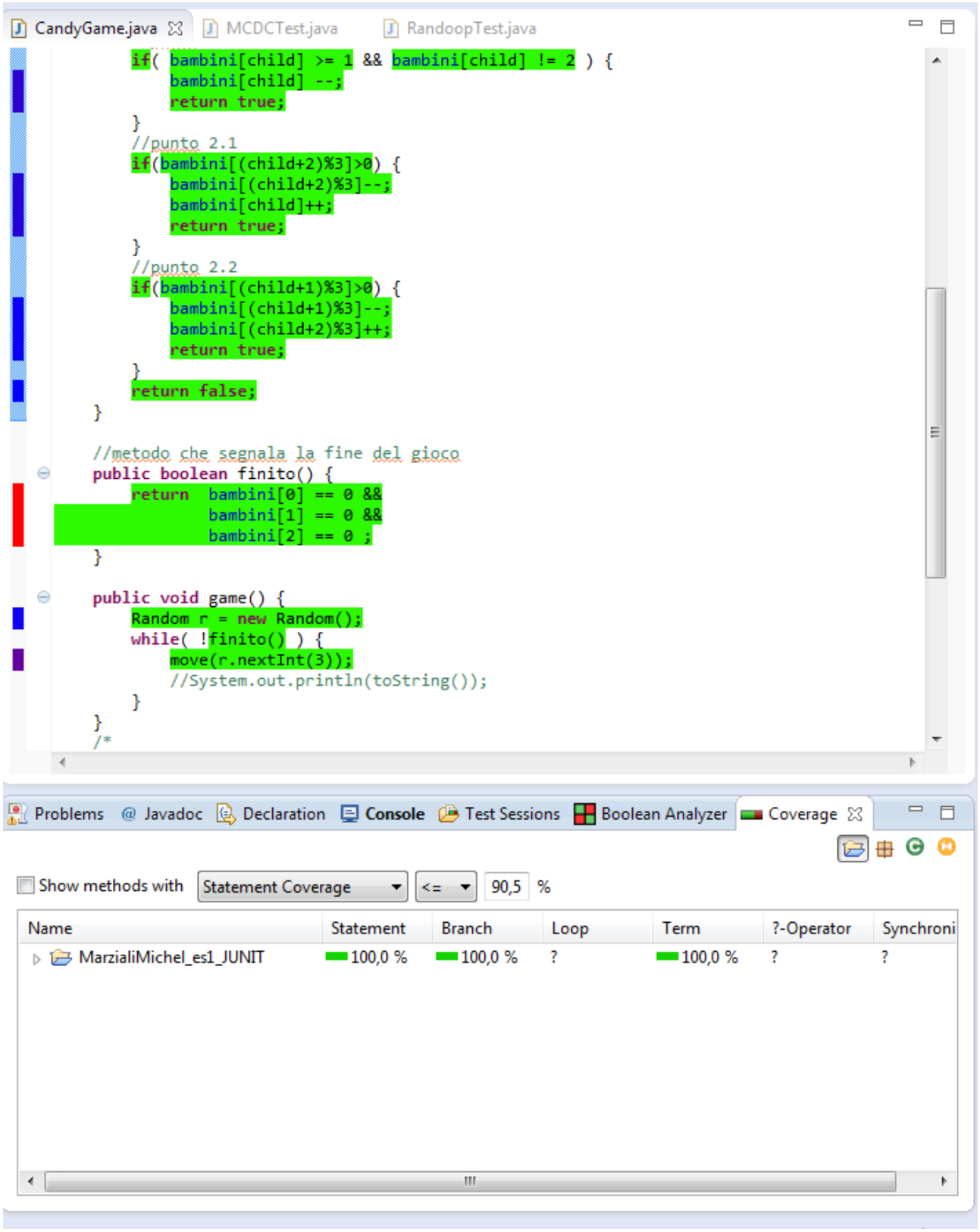
Problems | Javadoc | Declaration | Console | Test Sessions | Boolean Analyzer | Coverage

Test Session Container: MarzialiMichel_es1_JUNIT 30-giu-2015 15.47.43

Name	Date	Time
30-giu-2015 15.47.49	30-giu-2015	15.47.49
RandoopTest0:test100	30-giu-2015	15.47.48
RandoopTest0:test101	30-giu-2015	15.47.48
RandoopTest0:test102	30-giu-2015	15.47.48
RandoopTest0:test103	30-giu-2015	15.47.48
RandoopTest0:test104	30-giu-2015	15.47.48
RandoopTest0:test105	30-giu-2015	15.47.48
RandoopTest0:test106	30-giu-2015	15.47.48
RandoopTest0:test107	30-giu-2015	15.47.48
RandoopTest0:test108	30-giu-2015	15.47.48

Testing & Verifica del SW

Esame del 20/06/2015 di Marziali Michel



Testing & Verifica del SW

Esame del 20/06/2015 di Marziali Michel

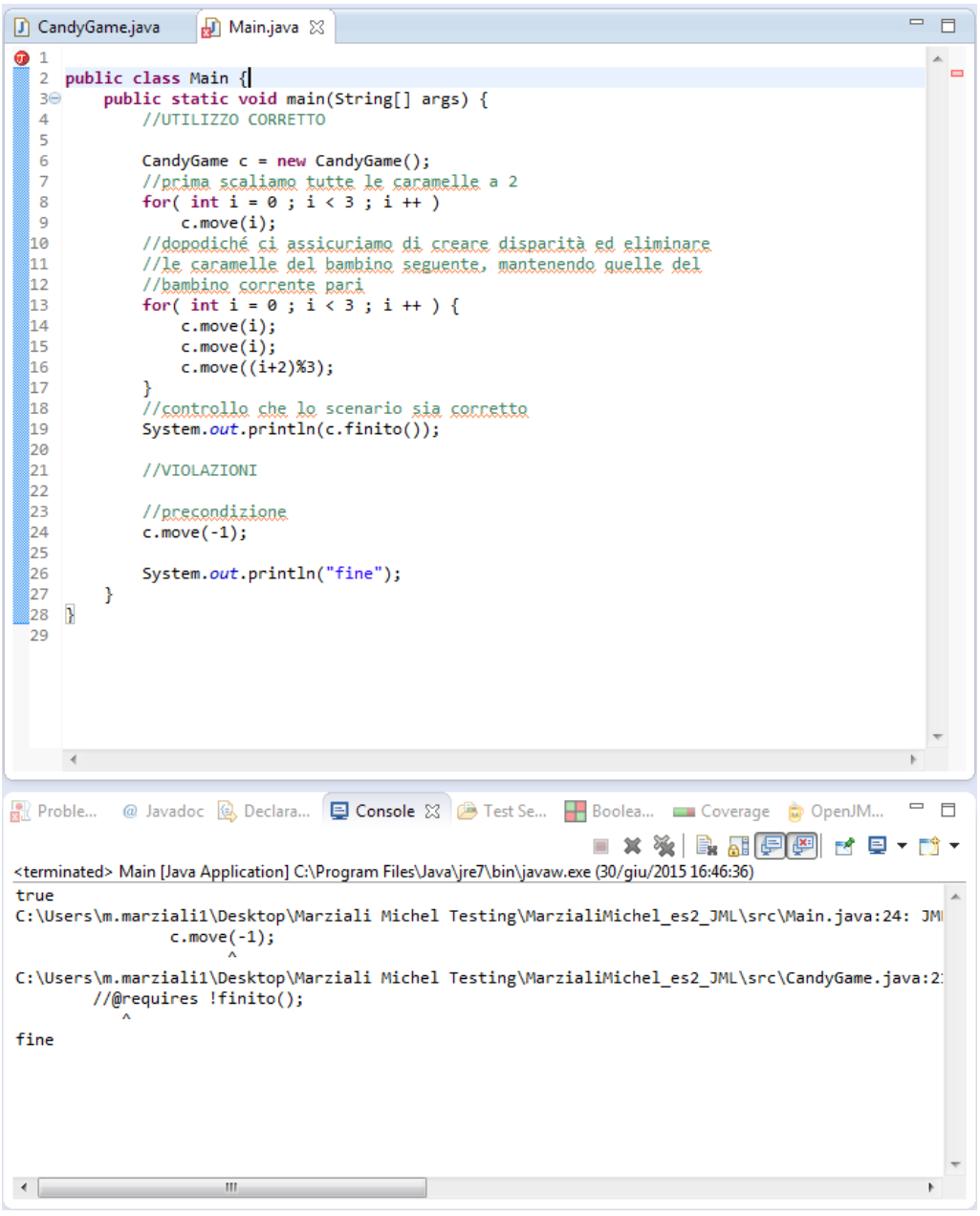
Esercizio 2

Nell'esercizio 2, converto tutte le condizioni date dagli if all'interno dei metodi in contratti che regolino il corretto andamento del programma. Per comodità ho lasciato le istruzioni definite nei contratti anche nelle funzioni, in modo da poterli violare senza incappare in eccezioni di Java.

Aggiungo come invariant che il sistema non possieda mai più di 9 caramelle al suo interno e che l'array sia sempre definito (non null).

Come postcondizioni pretendo che il costruttore inizializzi correttamente il sistema, che il move ritorni true solo nel caso in cui sia avvenuto un cambiamento, che la condizione di finito sia corretta e che game possa terminare solo se finito si verifica.

Dopo aver trascritto i contratti eseguiamo prima un test funzionante eseguendo il percorso previsto dalla classe di test precedentemente scritta CandyGameTest, dopodichè proviamo a violare precondizioni e postcondizioni con successo:



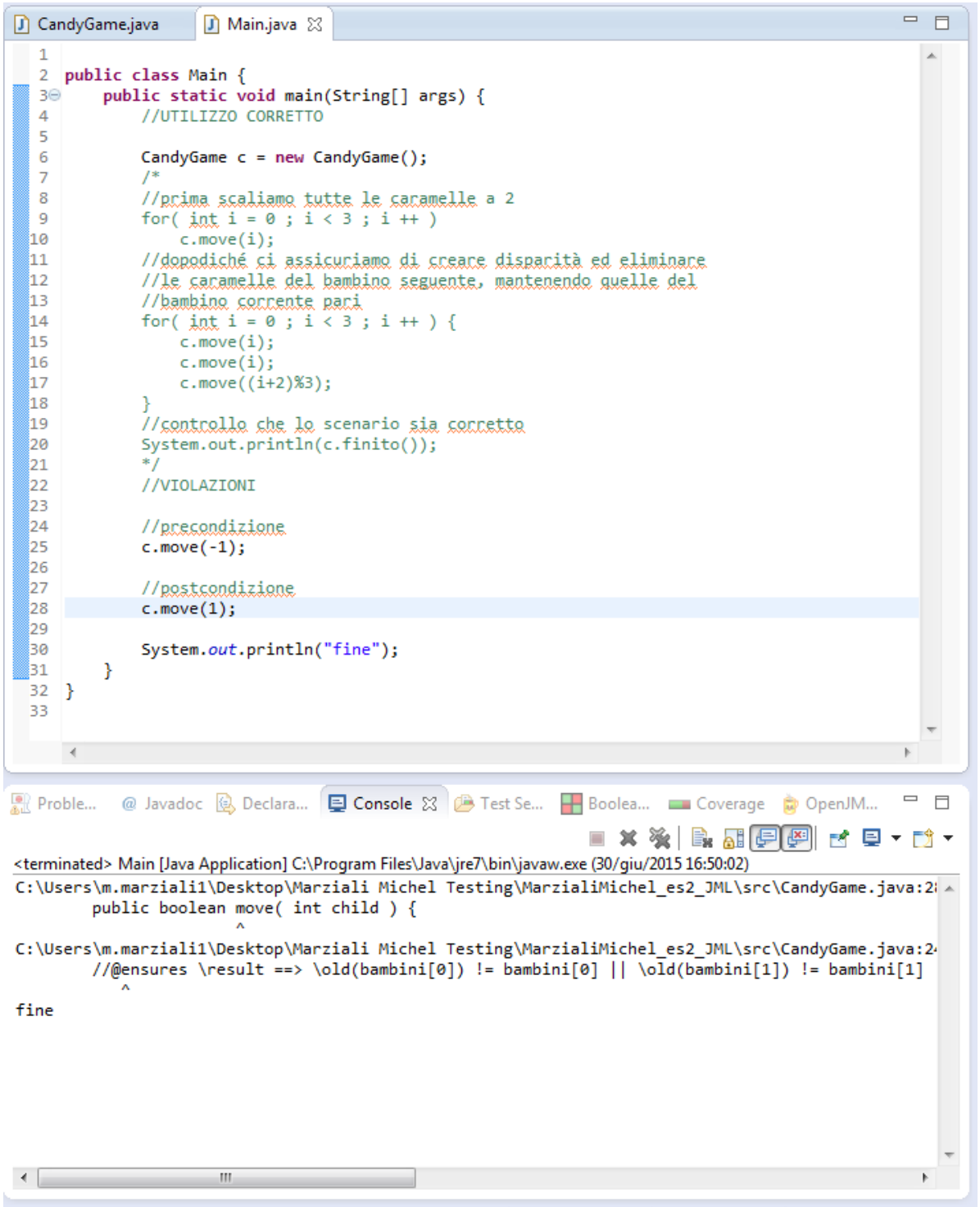
Testing & Verifica del SW

Esame del 20/06/2015 di Marziali Michel

Per poter violare una postcondizione introduciamo un errore nel codice:

```
public boolean move( int child ) {
    if(finito() || child<0 || child >= 3)
        return false;
    if( bambini[child] >= 1 && bambini[child] != 2 ) {
        //bambini[child] --;
        return true;
    }
```

Con il seguente risultato:

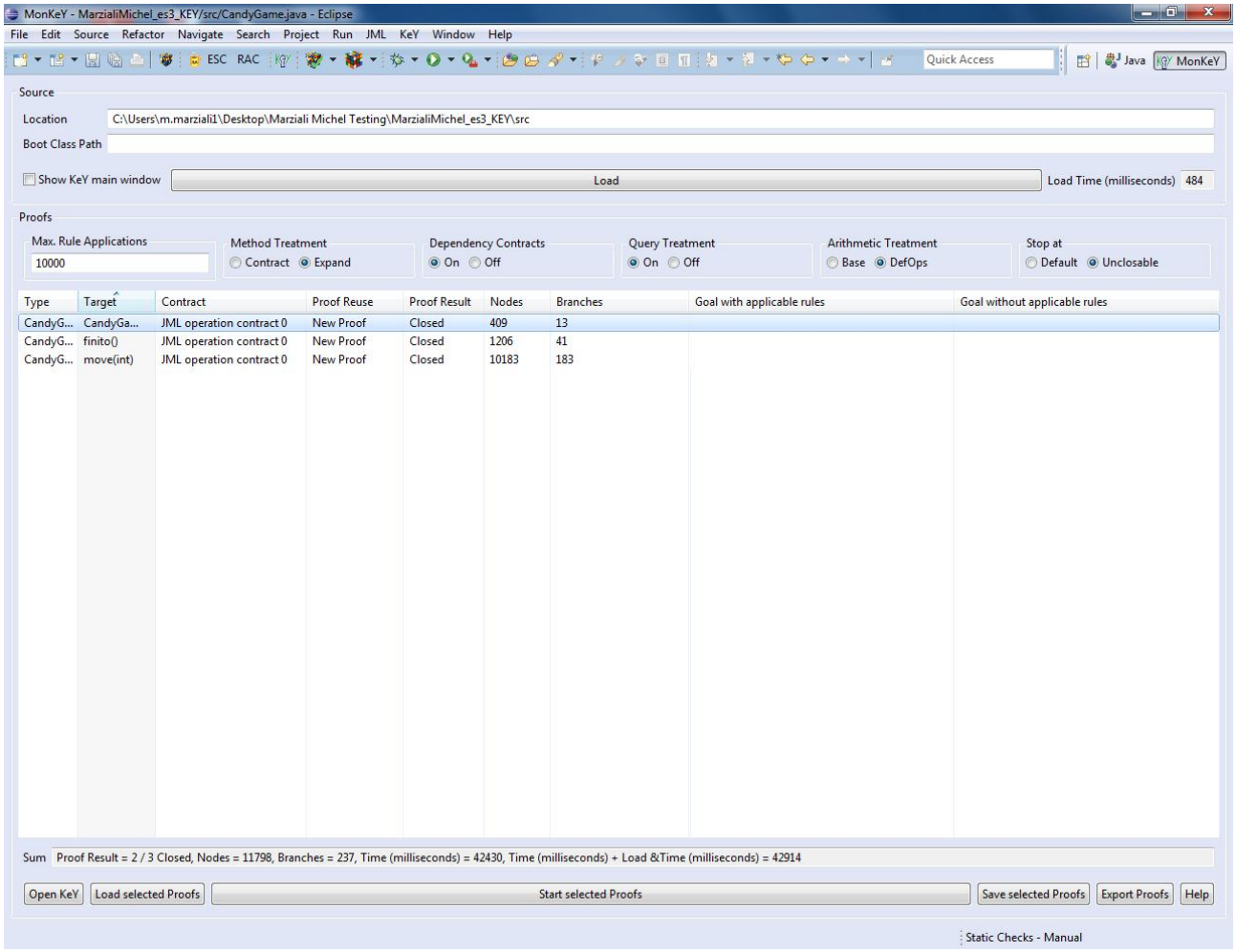


Testing & Verifica del SW

Esame del 20/06/2015 di Marziali Michel

Esercizio 3

Dopo aver scritto i contratti necessari al corretto funzionamento del programma nell'esercizio precedente li testiamo con Key.



E' stata omesso il metodo "game" dato che prevedeva l'utilizzo della classe esterna Random.

Il risultato ottenuto è soddisfacente dato che ogni metodo viene coperto correttamente dai contratti scritti.

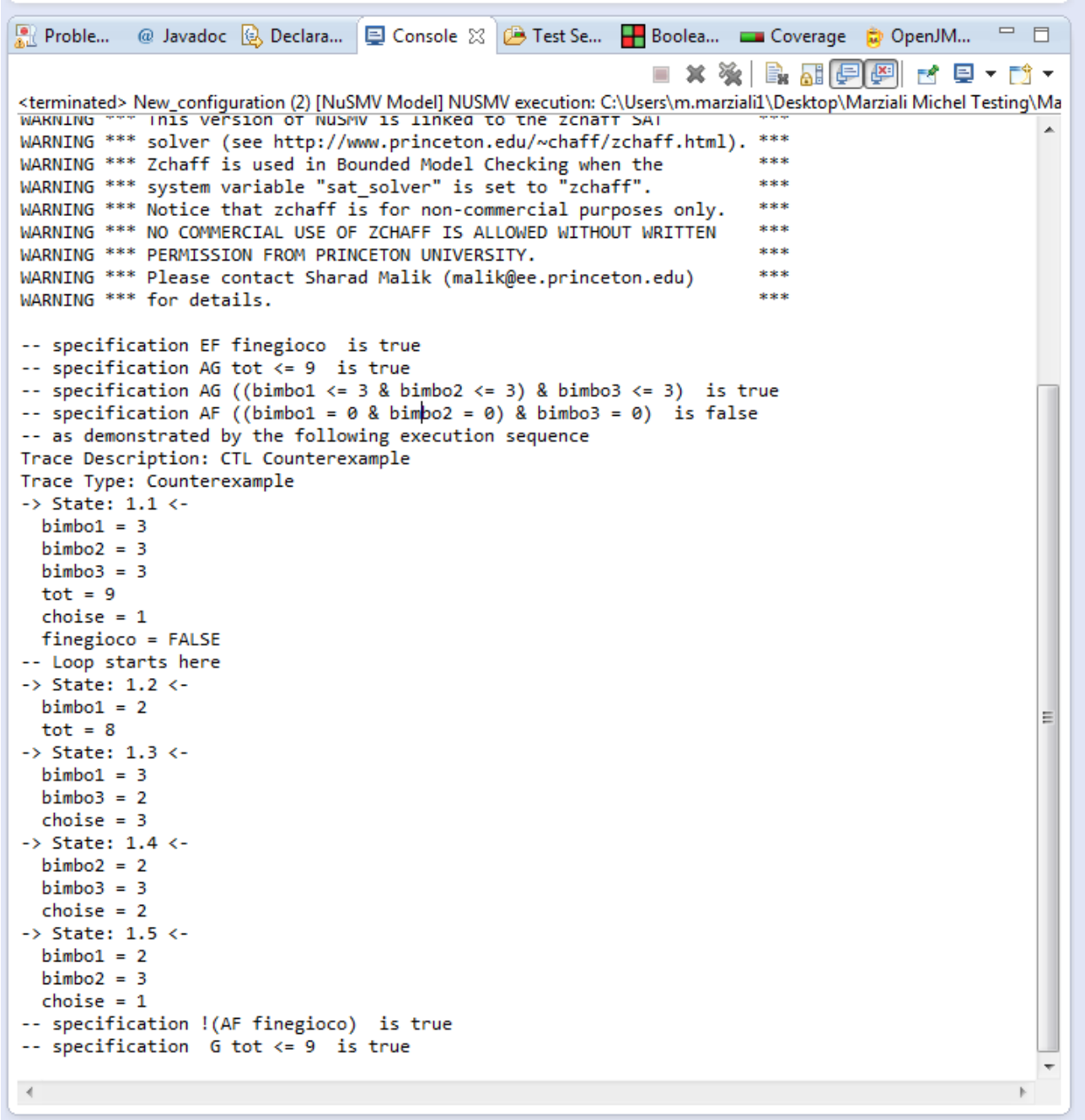
Testing & Verifica del SW

Esame del 20/06/2015 di Marziali Michel

Esercizio 4

Il quarto esercizio prevede l'analisi attraverso il tool NuSmv dei possibili stati attraversabili dal programma e la verifica di 4 condizioni.

Come possiamo notare dallo screen dei risultati la 1, la 2 e la 4 sono vere mentre la 3 non lo è, dunque viene riportato un controesempio.



Oltre alle specifiche richieste vengono inserite delle proprietà di safety e liveness.

La prima prevede che il sistema non abbia mai più di 9 caramelle in circolo, la seconda, invece, chiede che il programma possa giungere ad una conclusione.

Testing & Verifica del SW

Esame del 20/06/2015 di Marziali Michel

```
--liveness
CTLSPEC
	EF finegioco;

--safety
LTLSPEC
	G tot <= 9;
```

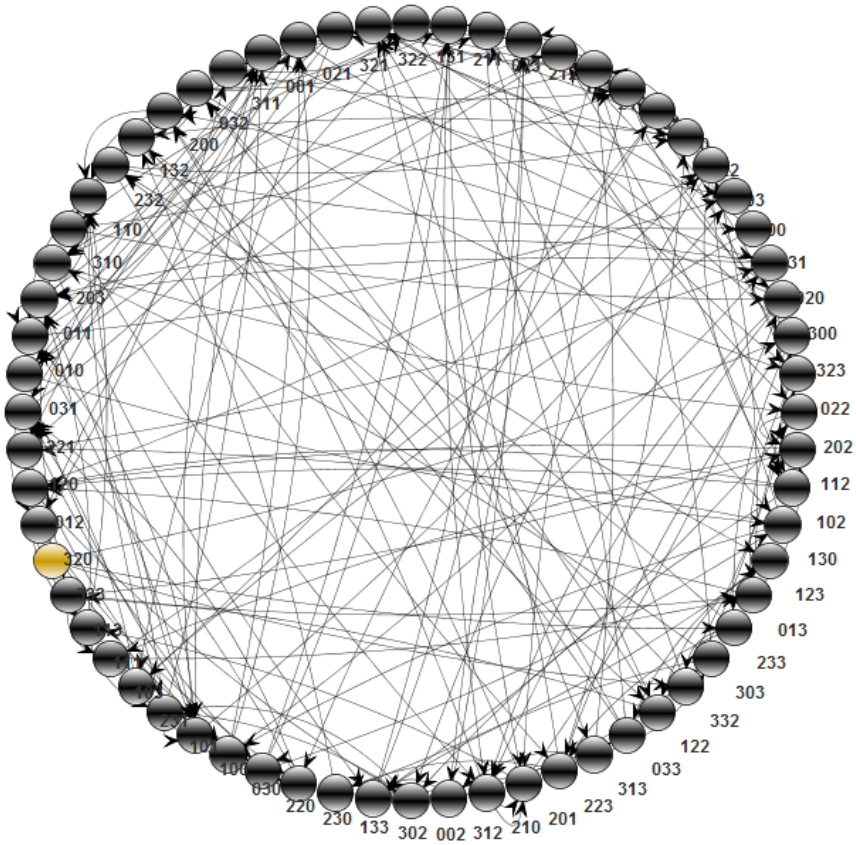
Testing & Verifica del SW

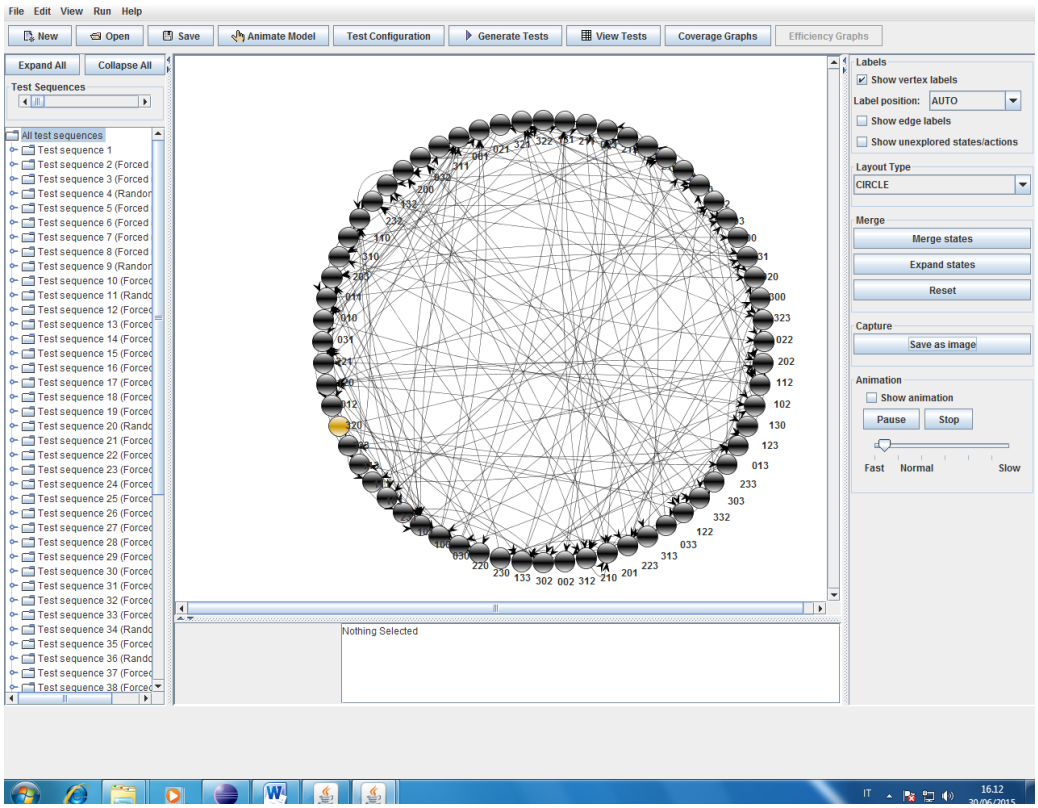
Esame del 20/06/2015 di Marziali Michel

Esercizio 5

Devo creare la FSM attraverso l'interfaccia Java FSMModel, per farlo riutilizzo i vecchi metodi mettendoli private e aggiungo dei metodi Guard che controllino il corretto inserimento dei dati.

Viene generata la FSM utilizzando ModelJUnit:

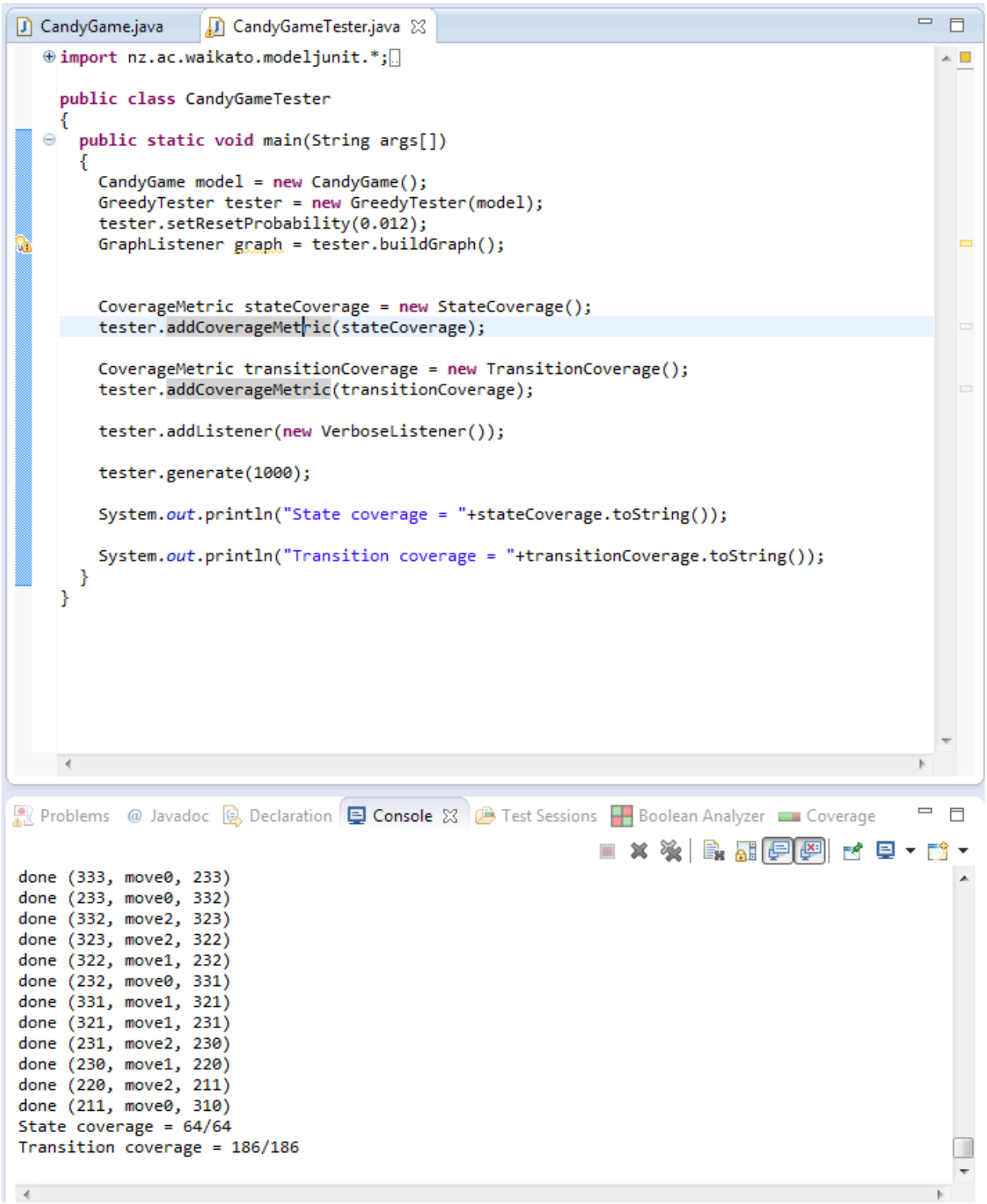




Testing & Verifica del SW

Esame del 20/06/2015 di Marziali Michel

Come metodo di generazione dei casi di test è stato scelto un metodo Greedy di lunga durata e con 1000 casi di test.



```
import nz.ac.waikato.modeljunit.*;

public class CandyGameTester
{
  public static void main(String args[])
  {
    CandyGame model = new CandyGame();
    GreedyTester tester = new GreedyTester(model);
    tester.setResetProbability(0.012);
    GraphListener graph = tester.buildGraph();


    CoverageMetric stateCoverage = new StateCoverage();
    tester.addCoverageMetric(stateCoverage);

    CoverageMetric transitionCoverage = new TransitionCoverage();
    tester.addCoverageMetric(transitionCoverage);

    tester.addListener(new VerboseListener());

    tester.generate(1000);

    System.out.println("State coverage = "+stateCoverage.toString());

    System.out.println("Transition coverage = "+transitionCoverage.toString());
  }
}
```

```
done (333, move0, 233)
done (233, move0, 332)
done (332, move2, 323)
done (323, move2, 322)
done (322, move1, 232)
done (232, move0, 331)
done (331, move1, 321)
done (321, move1, 231)
done (231, move2, 230)
done (230, move1, 220)
done (220, move2, 211)
done (211, move0, 310)
State coverage = 64/64
Transition coverage = 186/186
```

Come possiamo notare otteniamo una copertura totale sia degli stati che delle transizioni.

Testing & Verifica del SW

Esame del 20/06/2015 di Marziali Michel

Esercizio 6

Bisogna ottenere, attraverso il combinatorial testing, tutti i casi possibili in cui può trovarsi il sistema dati 3 bambini con 3 caramelle massimo a testa, utilizzando la copertura pairwise e 3-wise.

Copertura Pairwise

CandyGame.citl

```
Model CandyGame

Parameters:
    Range bimbo1 [ 0 .. 3 ];
    Range bimbo2 [ 0 .. 3 ];
    Range bimbo3 [ 0 .. 3 ];
    Boolean giocoFinito;
end
Constraints:
    //condizione di fine gioco
    # bimbo1==0 && bimbo2==0 && bimbo3==0 => giocoFinito #
    # bimbo1!=0 || bimbo2!=0 || bimbo3!=0 => !giocoFinito #
end
```

Test	bimbo1	bimbo2	bimbo3	giocoFinito
1	0	0	0	true
2	0	1	3	false
3	0	2	2	false
4	0	3	1	false
5	1	0	1	false
6	1	1	0	false
7	1	2	3	false
8	1	3	2	false
9	2	0	2	false
10	2	1	1	false
11	2	2	0	false

Name: IpoF

Time: 0.156

16

Export

CandyGame IpoF 2015/06/30 16:21:04

Copertura 3-wise:

Testing & Verifica del SW

Esame del 20/06/2015 di Marziali Michel

CandyGame.citl

```
Model CandyGame

Parameters:
    Range bimbo1 [ 0 .. 3 ];
    Range bimbo2 [ 0 .. 3 ];
    Range bimbo3 [ 0 .. 3 ];
    Boolean giocoFinito;
end
Constraints:
    //condizione di fine gioco
    # bimbo1==0 && bimbo2==0 && bimbo3==0 => giocoFinito #
    # bimbo1!=0 || bimbo2!=0 || bimbo3!=0 => !giocoFinito #
end
```

Test	bimbo1	bimbo2	bimbo3	giocoFinito
1	0	0	0	true
2	0	0	1	false
3	0	0	2	false
4	0	0	3	false
5	0	1	0	false
6	0	1	1	false
7	0	1	2	false
8	0	1	3	false
9	0	2	0	false
10	0	2	1	false
11	0	2	2	false

Name: IpoF

Time: 0.39

64

Export

CandyGame IpoF 2015/06/30 16:21:48

Testing & Verifica del SW

Esame del 20/06/2015 di Marziali Michel

Conclusione

Altra documentazione per facilitare la correzione la si può trovare nei commenti all'interno del progetto.