Problema del rushhour

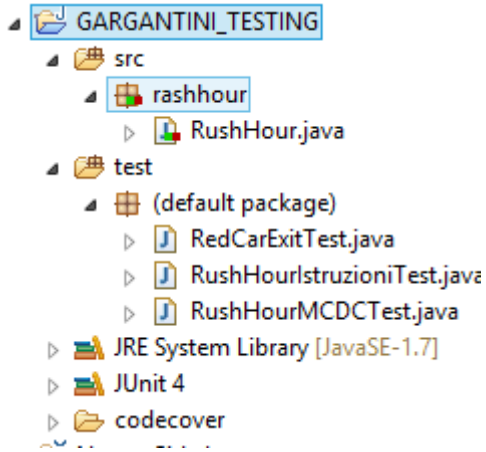
Soluzione proposta da Angelo Gargantini

TESTING

Spiego come ho fatto la struttura dati in Java (uso di grid etc.)

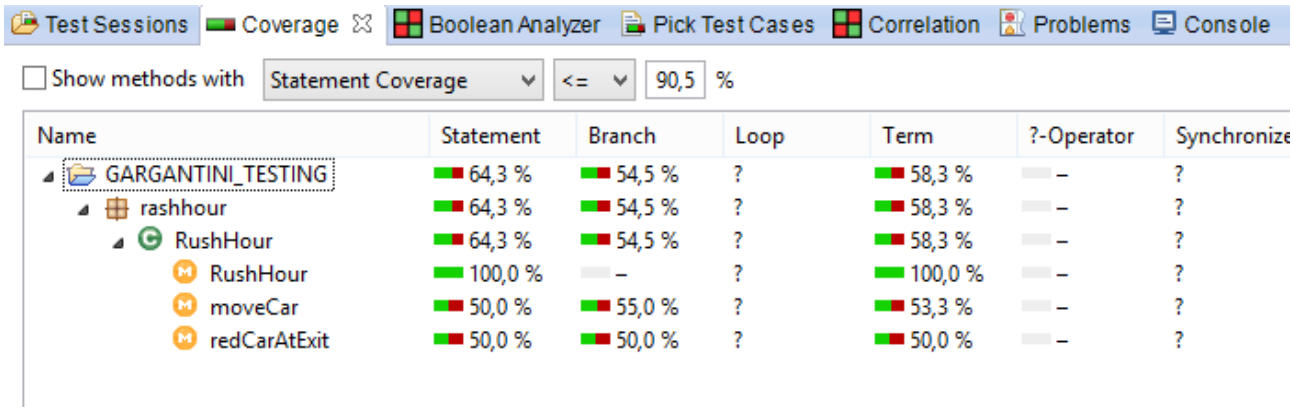
Scenario

Scrivo caso di test relativo allo scenario: NOTA BENE: metto i casi di test in un folder separato:




```
/**
 * The Class RedCarExitTest. Scenario in cui la macchina esce
 */
RedCarExitTest
```

Eseguo lo scenario e controllo la copertura (uso lo strumento di cattura per fare screenshots)

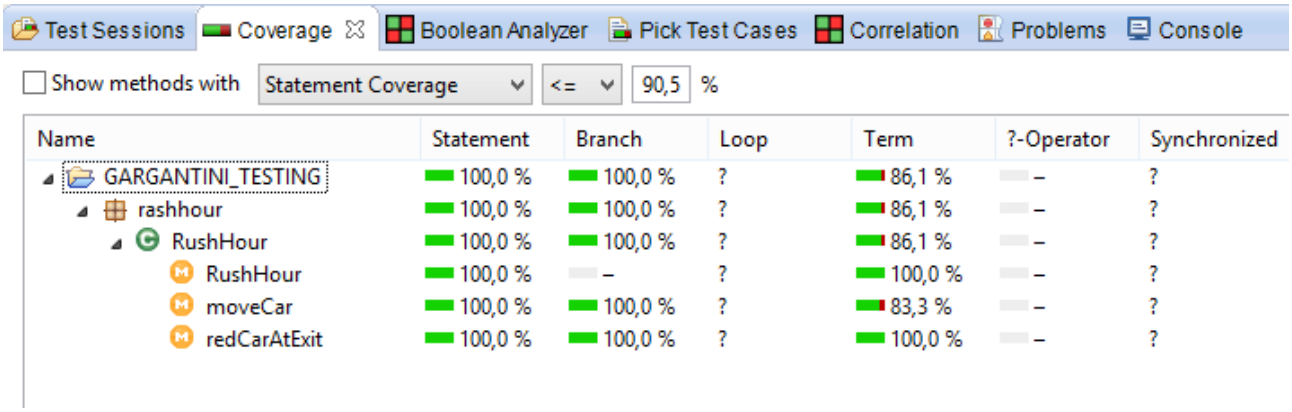


Come si vede la copertura non è del 100% neanche delle istruzioni.

Copertura istruzioni e branch

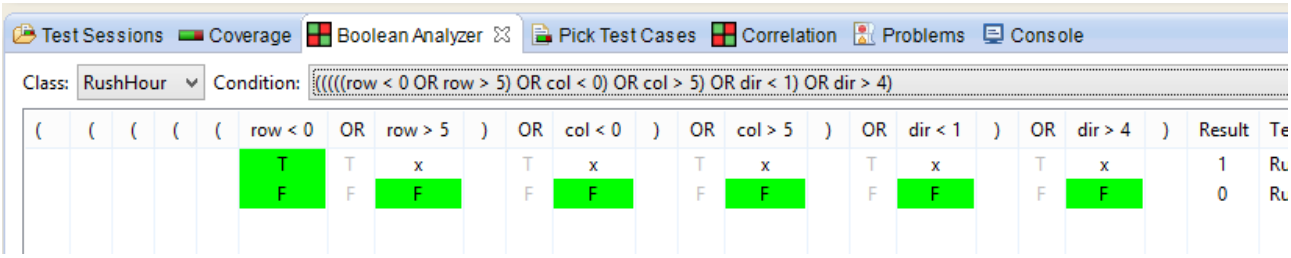
Aggiungo dei casi di test.

Scrivo un caso di test `RushHourIstruzioniTest` in cui aggiungo un può di movimenti non coperti dal test precedente. Ottengo la seguente copertura:



Copertura condizioni e MCDC

Ho una copertura degli statement e branch del 100%. Tuttavia se guardo la copertura delle condizioni nella decisione: `(row < 0 || row > 5 || col < 0 || col > 5 || dir < 1 || dir > 4)`



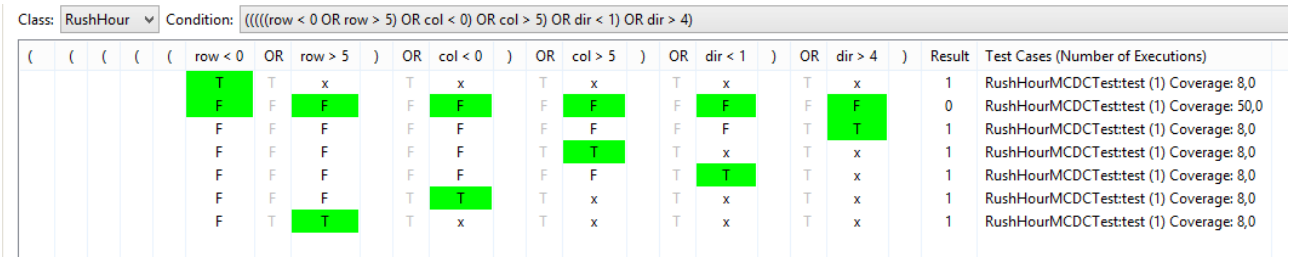
Mi accorgo che solo 2 casi di test ma molte condizioni non sono coperte (non ha neanche copertura delle condizioni) (mi accorgo anche dal colore delle condizioni:

```
if (row < 0 || row > 5 || col < 0 || col > 5 || dir < 1 || dir > 4) {
```

Scrivo un caso di test in cui faccio variare solo una condizione vera tenendo tutte le altre a false (essendo un OR) per ottenere la copertura MCDC. Il caso di test si chiama:

`RushHourMCDCTest`

Mi basta in questo test chiamare moveCar con un indice sbagliato e gli altri giusti. Ottengo la copertura:



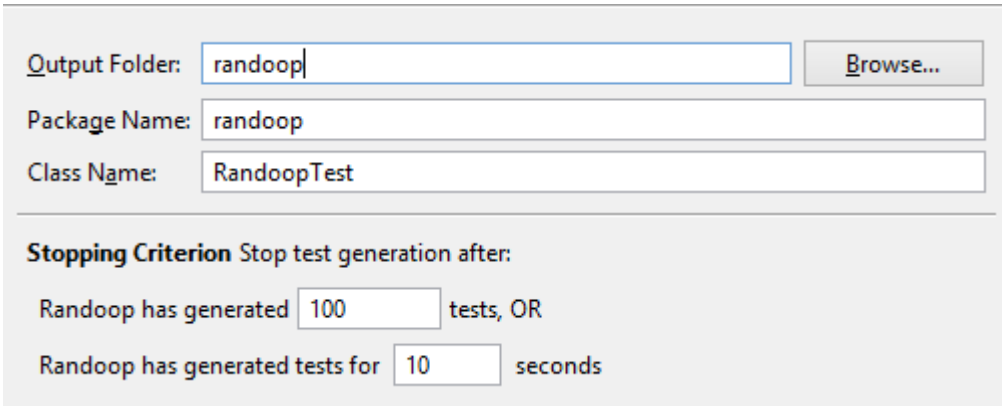
Che mi permette di coprire tutte le condizioni:

`(row < 0 || row > 5 || col < 0 || col > 5 || dir < 1 || dir > 4)`

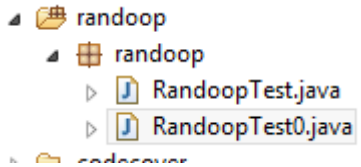
E anche l'MCDC.

Randoop

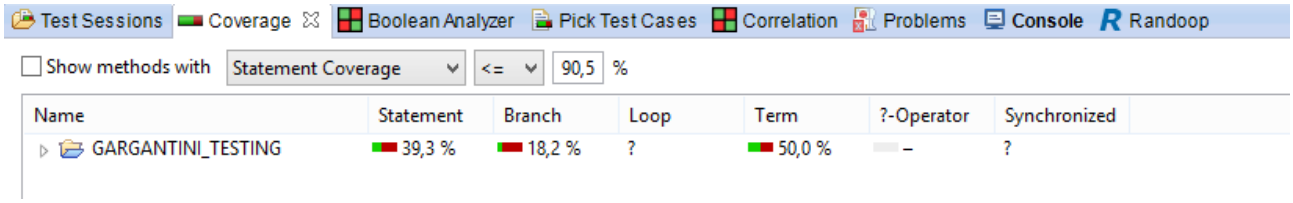
Genero casi di test con randoop. Attenzione abbasso il numero di test e i secondi per fare prima (altrimenti mi blocca per troppo tempo):



Ho generato i casi di test con randoop (un po' pochi – circa 100 - ma sicuramente non di meno di quelli che ho scritto a mano)



Che adesso lancio per valutare la copertura:



La copertura è molta bassa – come atteso.

JML

Creo nuovo progetto e copio la classe. Modifica la visibilità del campo:

```
/*@ spec_public @*/ int griglia[][];
```

Aggiungo i contratti:

Per il metodo move chiedo:

che gli indici siano giusti e che la posizione da muovere non sia vuota:

```
//@ requires !(row < 0 || row > 5 || col < 0 || col > 5 || dir < 1 || dir > 4);
//@ requires gameBoard[row][col] != 0;
```

Poi garantisco che se muovo la cella diventa vuota e viceversa se non muovo la cella rimane occupata:

```
//@ ensures \result ==> gameBoard[row][col] == 0;
//@ ensures !\result ==> gameBoard[row][col] != 0;
```

Potrei anche aggiungere post condizioni più complesse che dicono meglio quando sposto

```
@ ensures (!(\result) && \old(gameBoard [row][col])== gameBoard [row][col])||((\result)
&& gameBoard [row][col]==0);
```

```
@ ensures \result <==>((dir==4 &&\old(gameBoard [row][col]== gameBoard [row][col-
1]))||(dir==1 &&\old(gameBoard [row][col]== gameBoard [row-1][col]))||(dir==2
&&\old(gameBoard [row][col]== gameBoard [row][col+1]))||(dir==3 &&\old(gameBoard
[row][col]== gameBoard [row+1][col]));
```

Potrei anche aggiungere un po' di post condizioni al costruttore tipo:

```
/*@ requires true;
  @ ensures gameBoard[2][2]==1;
  @*/
```

E anche al metodo `redCarAtExit`

```
      /*@ requires true;
       @ ensures (\result <==> gameBoard[2][5]==1);
       @*/
```

Poi aggiungo diversi invarianti (non sono al 100% sicuro)

```
// nella griglia ci sono sempre 30 celle vuote
//@ public invariant (\num_of int i, j; i >=0 && i < 6 && j >= 0 && j < 6;
gameBoard[i][j] == 0) == 30;

// le macchine sono 6
//@ public invariant (\num_of int i, j; i >=0 && i < 6 && j >= 0 && j < 6;
gameBoard[i][j] != 0) == 6;

// la somma fa sempre 21
//@ public invariant  (\sum int i, j; i >=0 && i < 6 && j >= 0 && j < 6;
gameBoard[i][j]) == 21;

// non esiste una cella con valore maggiore di 6
//@ public invariant !(\exists int i, j; i >=0 && i < 6 && j >= 0 && j < 6;
gameBoard[i][j] > 6);

// tutte le celle hanno valore maggiore o uguale a 0
//@ public invariant (\forall int i, j; i >=0 && i < 6 && j >= 0 && j < 6;
gameBoard[i][j] >=0);
```

Quando testo la mia classe, posso semplicemente violare le precondizioni:

```
rh.moveCar(-1, 0, 3);
```

→

```
JML precondition is false
      //@ requires !(row < 0 || row > 5 || col < 0 || col > 5 || dir < 1 || dir > 4);
          ^
```

Oppure posso anche modificare il codice in modo che venga violate una postcondizione.

Ad esempio se commento nel metodo move:

```
//gameBoard[row][col] = 0;
```

Ho una volazione della postcondizione

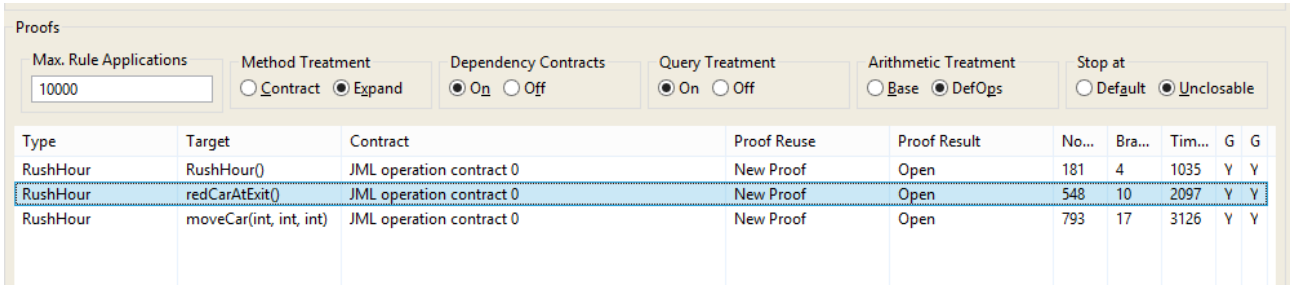
```
JML postcondition is false
            rh.moveCar(2, 2, 3);
                       ^
RushHour.java:52: Associated declaration:
ProvaRushHour.java:8:     //@ ensures \result ==> gameBoard[row][col] == 0;
```

^

KEY

Copio la classe da JML.

Provo a dimostrare. Tutto rimane open:



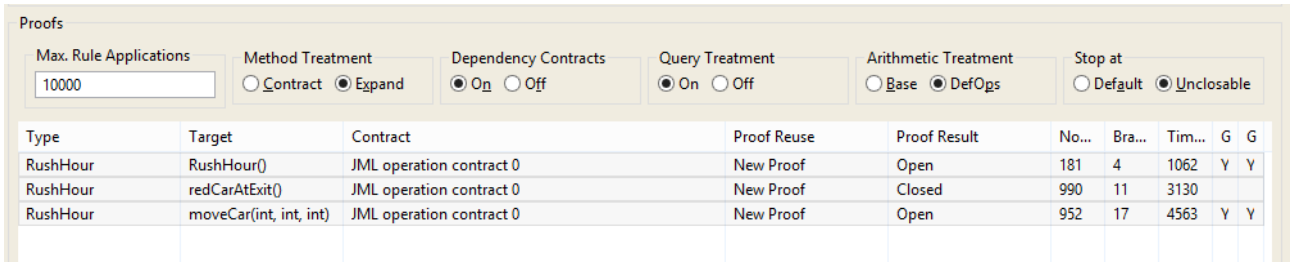
Type	Target	Contract	Proof Reuse	Proof Result	No...	Bra...	Tim...	G	G
RushHour	RushHour()	JML operation contract 0	New Proof	Open	181	4	1035	Y	Y
RushHour	redCarAtExit()	JML operation contract 0	New Proof	Open	548	10	2097	Y	Y
RushHour	moveCar(int, int, int)	JML operation contract 0	New Proof	Open	793	17	3126	Y	Y

Guardando il fallimento capisco che mancano delle condizioni sull'array:

Aggiungo:

```
//@ public invariant gameBoard.length == 6;
//@ public invariant gameBoard[2].length == 6;
```

E a questo punto chiudo almeno un metodo:



Type	Target	Contract	Proof Reuse	Proof Result	No...	Bra...	Tim...	G	G
RushHour	RushHour()	JML operation contract 0	New Proof	Open	181	4	1062	Y	Y
RushHour	redCarAtExit()	JML operation contract 0	New Proof	Closed	990	11	3130		
RushHour	moveCar(int, int, int)	JML operation contract 0	New Proof	Open	952	17	4563	Y	Y

NUMSV

GRIGLIA 6x6 MA SOLO 3 MACCHINE: 1, 2 e 3.

In questo caso uso solo 3 macchine che rappresento con row e colum:

```
MODULE car(carNum,c1,c2,moveCar,dir)
VAR
  row : 1..6;
  col : 1..6;
```

in questo modo posso dare la regola generale per una macchine per muoversi (vedi codice)

Provo le seguenti proprietà:

```
-- non è mai possibile avere 2 macchine nello stesso posto
-- car1 e car2
CTLSPEC !EF(car1.row=car2.row & car1.col=car2.col);
-- car1 e car3
CTLSPEC !EF(car1.row=car3.row & car1.col=car3.col);
```

```
-- car3 e car2
CTLSPEC !EF(car2.row=car3.row & car2.col=car3.col);
-- LIVENESS:
--  la macchine 3 può scendere
CTLSPEC EF(car3.row=4 & car3.col=6);
--  la macchine rossa può uscire
CTLSPEC EF (car1.row=3 & car1.col=6);
```

Se voglio trovare come fare uscire la machina posso cercare di provare che la macchina non esce mai:

```
CTLSPEC AG ! (car1.row=3 & car1.col=6);
```

E trovo il seguente comportamento:

```
-> State: 1.1 <-
  dir = UP
  moveCar = 1
-> State: 1.2 <-
  dir = DOWN
  car1.row = 2
-> State: 1.3 <-
  moveCar = 3
  car1.row = 3
-> State: 1.4 <-
  dir = RIGHT
  moveCar = 1
  car3.row = 4
-> State: 1.5 <-
  car1.col = 4
-> State: 1.6 <-
  car1.col = 5
-> State: 1.7 <-
  dir = UP
  car1.col = 6
```

FSM

Scrivo un modello FSM per la versione semplificata.

Metto una azione per ogni macchina e ogni direzione con le opportune guardie. Ad esempio per spostare le macchina 1 a sinistra:

```
	@Action
	public void left1() {
		car1.col--;
	}
```

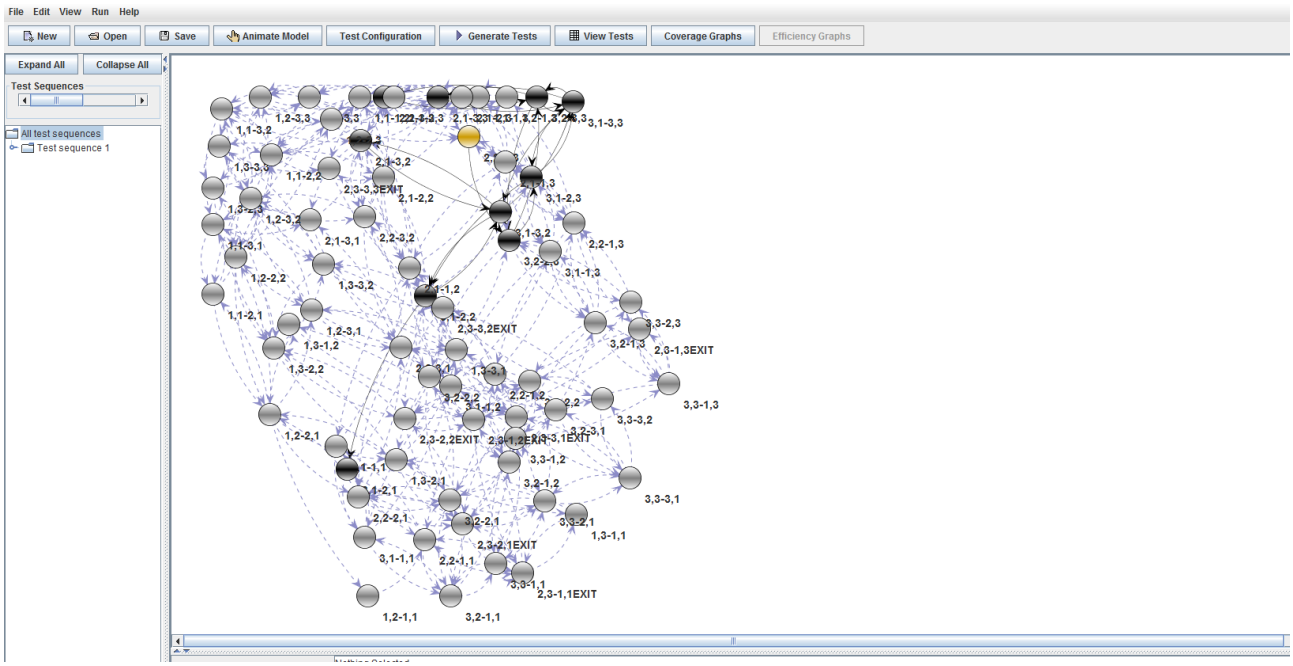
Con la guardia che può andare a sinistra

```
	public boolean left1Guard() {
		return (car1.col > 1)
				&& !(car2.row == car1.row && car2.col == car1.col - 1);
	}
```

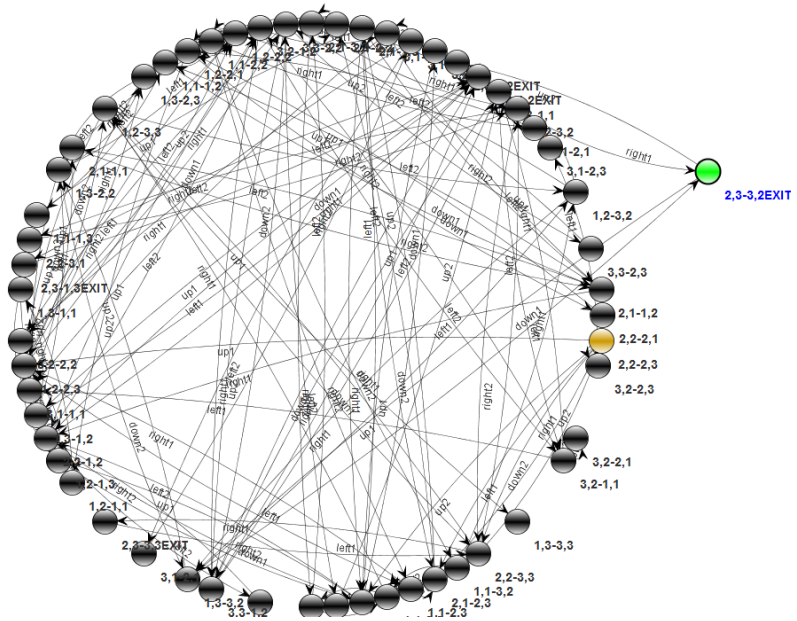
Metto anche lo state che mi dice la posizione delle due macchine.

Provo a generare la FSM (esporto jar, carico progetto, etc.)

I primi tentativi mi danno risultati poco leggibili:



Lo sistemo un po’ fino ad ottenere:



Nota che il numero di stati raggiungibili è comunque elevato: sono 9 posizioni per la macchina 1 e 8 per la macchina 2, quindi 72.

[Alternativamente potevo riusare la classe del testing e mandare azioni a caso]