

# TESTING E VERIFICA DEL SOFTWARE

## ESERCITAZIONE 4 – JUnit e CodeCover

### INDICAZIONI PER UTILIZZARE JUnit/CodeCover

Assicurati di avere installato CodeCover

Crea un progetto Java contenente la classe/le classi da testare.

Per effettuare i test, creare un nuovo JUnit Test Case (new/other/Java/JUnit/Junit Test Case). Nei casi di test JUnit usa gli opportuni assert.

Per abilitare CodeCover e i criteri di coverage vai sulla voce Project/Properties/CodeCover, abilita CodeCover e spunta le diverse voci.

Per misurare la copertura, clicca tasto destro sulla classe (o package) che si vuole coprire e spuntando la voce "Use For Coverage Measurement".

Lancia i tuoi casi di test con Run as JUnit.

Per vedere la copertura usa la prespective "CodeCover". Per MCDC usa la view "Boolean Coverage"

Per qualche esercizio prova a generare i casi di test con Randoop (e Evosuite) e prova a valutare la copertura. Sei riuscito a battere randoop?

### Esercizio 1

Data la classe Rettangolo, creare una classe di Test nella quale si utilizzino i vari Assert.

```
public class Rettangolo {  
  
    private int base;  
    private int altezza;  
  
    public Rettangolo(int b, int a){  
        base=b;  
        altezza=a;  
    }  
    public int getBase(){           return base; }  
  
    public int getAltezza(){        return altezza;    }  
  
    public int getArea(){           return base * altezza;    }  
  
    public int getPerimetro(){      return 2 * (base + altezza);    }  
  
}
```

### Esercizio 2

Data la classe Light, creare una classe di Test che soddisfi i criteri di copertura di istruzioni, rami (branch), condizioni e MCDC, facendo variare i valori dei 3 parametri booleani.

```

public class Light {

    private boolean lightOn=false;
    private boolean bottomIn=false;
    private boolean bottomOut=false;

    Light(){}

    public boolean onOff(boolean light, boolean in, boolean out){

        this.bottomIn=in;
        this.bottomOut=out;

        if((bottomIn || bottomOut) && !light)
        {
            lightOn=true;
            return true; //luce accesa
        }
        else
        {
            lightOn=false;
            return false; //luce spenta
        }
    }
}

```

### Esercizio 3

Data la classe computer, scrivere un test set che soddisfi il criterio di copertura degli archi (branch coverage), scrivere poi un secondo test set che soddisfi il criterio MCDC.

```

public class Computer {

    boolean compute(boolean x, boolean y, boolean z, int temp, int thresh) {
        boolean no_alarm;
        if (temp <= thresh) {
            no_alarm = true;
        } else {
            no_alarm = false;
        }
        if ((x && (y || z)) && no_alarm)
            return true;
        else
            return false;
    }
}

```

### Esercizio 4

Data la classe Balance, creare una classe di Test che soddisfi i criteri di copertura di istruzioni, rami (branch), condizioni e MCDC, facendo variare i valori dei 2

parametri interi. Privare poi ad utilizzare gli Assert sulla classe Balance, sfruttando i metodi getP() e getW().

```
public class Balance {

    private int p;
    private int w;

    Balance(){}

    public boolean setBalanceValue(int x, int y){
        this.p=x;
        this.w=y;

        if((p<4 && w<=400) || (p==0 && w<=1000))
            return true;
        else
        {
            return false;
        }
    }

    public int getP(){
        return p;
    }

    public int getW(){
        return w;
    }
}
```

### Esercizio 5 (tema d'esame 27/01/15):

Un magazzino contiene **cinque** tipi di prodotti. Il magazzino può contenere al massimo **100** unità di ogni prodotto. Si possono aggiungere quantità di un certo prodotto ma al massimo **10** alla volta e solo fino a raggiungere il livello massimo.

In Java scrivi una classe Magazzino che implementa il sistema sopra. Ha tre metodi:

- boolean insert(int productIndex, int addQuantity)

dati un indice di prodotto productIndex e una quantità addQuantity, aggiunge il valore addQuantity (al massimo 10 prodotti); al prodotto identificato da productIndex Il metodo ritorna true se l'aggiunta viene eseguita, false altrimenti. L'aggiunta non viene eseguita se l'indice di prodotto è errato, o se la quantità aggiunta non è corretta (non compresa tra 1 e 10), o se la nuova quantità che si otterrebbe con l'aggiunta supera le 100 unità.

- boolean isFull(int productIndex)

dice se un certo prodotto ha raggiunto il massimo

- boolean isFull()

restituisce se il magazzino è completamente pieno (tutti i prodotti sono la massimo)

Le quantità di prodotti a magazzino sono memorizzate in un array di interi, dove ogni cella corrisponde ad un prodotto. All'inizio il magazzino è completamente vuoto.

□ Scrivere l'implementazione e i casi di test con Junit. Scrivi una classe di test

FullTest per lo scenario in cui a partire il magazzino si riempie del tutto. Esegui il controllo della copertura (istruzioni, branch, decisioni, MCDC). Se il caso sopra non è sufficiente ad avere una copertura soddisfacente, aggiungi i casi di test necessari (in una classe separata per ogni copertura).

NB: all'esame dovreste scrivere anche l'implementazione java del magazzino, qui per non perdere troppo tempo ve la forniamo.

```
public class Magazzino {

    private int[] product = new int[5]; // rappresenta i 5 prodotti, da 0 a 4

    public Magazzino() {
        for (int i = 0; i < 5; i++) {
            product[i] = 0;
        }
    }

    /**
     *
     * @param productIndex
     *     l'indice del prodotto al quale aggiungere addQuantity
     * @param addQuantity
     *     la quantità da aggiungere al prodotto productIndex
     * @return true se l'inserimento è avvenuto, false altrimenti
     */
    public boolean insert(int productIndex, int addQuantity) {
        if ((productIndex < 0 || productIndex > 4)
            || (addQuantity < 0 || addQuantity > 10)
            || (product[productIndex] + addQuantity) > 100) {
            return false;
        } else {
            product[productIndex] += addQuantity;
            return true;
        }
    }

    /**
     *
     * @param productIndex
     *     l'indice del prodotto che voglio controllare
     * @return se 'indice non è corretto o il magazzino non è pieno ritorno
     *         false, true altrimenti
     */
    public boolean isFull(int productIndex) {
        if (productIndex < 0 || productIndex > 4)
            return false;
        else
            return product[productIndex] == 100 ? true : false;
    }

    /**
     *
     * @return true se il magazzino è pieno, false altrimenti
     */
    public boolean isFull() {
```

```

    int total = 0;
    for (int i = 0; i < 5; i++)
        total += product[i];
    return total == 500 ? true : false;
}
}

```

### **Esercizio 6 (tema d'esame 21/07/14):**

Si vuole scrivere un modello e una implementazione semplificata per il puzzle RushHour. Nel RushHour una griglia 6x6 contiene 6 macchine numerate da 1 a 6 (ogni macchina per semplicità ha dimensione 1x1). Puoi pensare di rappresentare la griglia come array bidimensionale di interi con le celle vuote indicate con 0 oppure rappresentare la posizione delle 6 macchine con una coppia di interi da 0 a 5 per ogni macchina.

	1	2	3	4	5	6
1				6		
2						2
3			1			3
4						4
5		5				
6						

Obiettivo del gioco è portare la macchina numero 1 (chiamata macchina rossa) all'uscita della griglia, che corrisponde alla cella con indici (3,6) (o 2,5 se le coordinate iniziano da 0) indicata dallo sfondo scuro. Quando la macchina rossa raggiunge l'uscita il gioco termina.

La configurazione iniziale del gioco è quella indicata.

Il sistema deve permettere lo spostamento di una macchina in una casella adiacente libera. Deve inoltre essere possibile sapere se la macchina rossa è all'uscita.

□ Scrivere l'implementazione e i casi di test con Junit. Scrivi una classe di test RedCarExitTest per lo scenario in cui a partire dalla configurazione iniziale la macchina rossa esce. Esegui il controllo della copertura (istruzioni, branch, decisioni, MCDC). Se il caso sopra non è sufficiente ad avere una copertura soddisfacente, aggiungi i casi di test necessari (in una classe separata per ogni copertura).

```

public class RushHour {

    int griglia[][];

    public RushHour() {

        griglia = new int[6][6];

        // questo sarebbe inutile perchè è inizializzato a zero di default
        for (int i = 0; i <= 5; i++) {
            for (int j = 0; j <= 5; j++) {

```

```

        griglia[i][j] = 0;
    }
}

griglia[2][2] = 1; // macchina rossa
griglia[1][5] = 2;
griglia[2][5] = 3;
griglia[3][5] = 4;
griglia[4][1] = 5;
griglia[0][3] = 6;

}

/**
 * Move car.
 *
 * @param row la colonna della macchina da spostare: può essere un numero qualsiasi ma viene fatto il controllo che sia tra 0 e 5 compresi
 * @param col the col
 * @param dir la direzione: 1. verso l'alto; 2. verso destra; 3. verso il basso; 4. verso sinistra.
 * @return true, if successful la macchina è spostata
 */
public boolean moveCar(int row, int col, int dir) {
    if (row < 0 || row > 5 || col < 0 || col > 5 || dir < 1 || dir > 4) {
        // in tal caso ho sbagliato gli indici oppure ho indicato una direzione inesistente
        return false;
    }
    if (griglia[row][col] == 0) {
        // in tal caso ho selezionato una casella vuota, non posso spostare macchine
        return false;
    } else {
        int nRow, nCol; // nuove coordinate della macchina
        if (dir == 1) {
            // verso l'alto
            if (row == 0)
                // voglio andare verso l'alto, ma sono al bordo
                return false;
            nRow = row - 1;
            nCol = col;
        } else if (dir == 2) {
            // verso destra
            if (col == 5)
                // voglio andare verso destra, ma sono al bordo
                return false;
            nCol = col + 1;
            nRow = row;
        } else if (dir == 3) {
            // verso il basso
            if (row == 5)
                // voglio andare verso il basso, ma sono al bordo
                return false;
            nRow = row + 1;
            nCol = col;
        } else {
            assert (dir == 4);
            // verso sinistra

```

```

        if (col == 0)
            // in tal caso voglio andare verso sinistra, ma sono
            // bordo
            return false;
        nCol = col - 1;
        nRow = row;
    }
    if (griglia[nRow][nCol] != 0) {
        // la cella di destinazione è già occupata, non posso
        // spostare la macchinina
        return false;
    } else {
        // la cella di destinazione è libera, sposto la macchina.
        griglia[nRow][nCol] = griglia[row][col];
        griglia[row][col] = 0;
        return true;
    }
}

/**
 * Red car at exit.
 *
 * @return true, se la macchina rossa è in uscita
 */
public boolean redCarAtExit() {
    if (griglia[2][5] == 1) {
        return true;
    }
    return false;
}
}

```

## Testing di un programma TE sett 22

Sia data la seguente classe in Java che rappresenta una lampadina che può essere spenta o accesa. La lampadina è alimentata a 12 V, se si setta la tensione tra 11 e 13 ed è spenta, la lampadina si accende. Se si applica una tensione <2, la lampadina si spegne.

```

public class Lampadina{

    boolean accesa = false;

    // setta il voltaggio, accende o spegne la lampadina
    // restituisce true se lo stato cambia
    boolean setVolts(float volt) {
        if (!accesa && (volt <=13 || volt >=11)){
            accesa = true;
            return true;
        }
        if (accesa && volt <= 2){
            accesa = false;
            return true;
        }
    }
}

```

```
        return false;
    }
}
```

Scrivere i casi di test per ottenere la copertura

- Delle istruzioni
- Delle decisioni
- Delle condizioni
- MCDC

Va spiegato bene perché si scelgono certi input o certe sequenze di istruzioni.

Se per qualche motivo una condizione o una combinazioni di condizioni non è copribile, spiega perché e ignorala.