APPIANI ANDREA - 1057683

Progetto da esame passato:

MAGAZZINO – 27 Gennaio 2015

CLASSE JAVA

Ho implementato le funzionalità del magazzino specificate nella consegna all'interno di una classe Java denominata "Magazzino.java" contenente i metodi:

- *Magazzino()* – il costruttore;
- *insert(int productIndex, int addQuantity)* – per l'incremento di quantità di un prodotto;
- *isFull(int productIndex)* – per il controllo sul singolo prodotto;
- *isFull()* – per il controllo sul magazzino, quindi su tutti i prodotti.

Tutti i metodi, eccetto il costruttore, ritornano un Boolean rappresentante l'esito dell'azione.

ANALISI STATICA

Un'analisi fatta con il tool PMD ha riportato i seguenti problemi/consigli:

1) Il parametro passato a "isFull(int)" potrebbe essere Final;
2) Commenti richiesti al campo "prodotti" e ai metodi "isFull()" e "isFull(int)" e costruttore;
3) Sostituire il classico ciclo "for" con un "for-each" nel metodo "isFull()";
4) Usare le parentesi graffe per racchiudere i corpi degli "if";
5) Il campo "prodotti" dovrebbe essere dichiarato "transient" (in quanto non sono implementati i metodi "get" e "set");
6) Sconsiglia l'uso di nomi brevi per variabili, ad esempio "i";

REFRACTORING

In seguito all'analisi statica sono state apportate le rispettive modifiche consigliate al codice per migliorarne la scrittura senza però modificarne l'esecuzione.

CODE INSPECTION CHECKLIST

Partendo da una checklist completa per ogni codice Java si sono selezionati solo i casi più significativi vista la semplicita del progetto in questione.

Il simbolo ☒ significa che il codice NON presenta errori di quel tipo.

1. ☒Variabili e costanti sono usate secondo le "naming conventions";
2. ☒ Variabili con nomi troppo simili e/o scritte con errori di battitura;
3. ☒ Tutte le variabili di controllo per cicli FOR sono dichiarate nella header del loop;
4. ☐ Tutti gli attributi/metodi hanno modificatori di visibilità appropriati;
5. ☐ Ogni parametro viene controllato prima di essere usato nel metodo;
6. ☐ Ogni classe ha un appropriato costruttore e distruttore;

7. ☒ Operazioni usanti data types diversi (ed errati);
8. ☒ E' possibile causare overflow con una o più computazioni;
9. ☒ Le parentesi sono usate per evitare ambiguità nel codice;
10. ☒ Ogni espressione booleana è corretta;
11. ☒ Ogni comparazione tra operatori è corretta;
12. ☒ Si ha un "&" inavvertitamente scambiato con un "&&", o un "|" in "||"?
13. ☒ Sono presenti "side-effects" derivanti da comparazioni booleane;
14. ☒ Per ogni loop è stato usato il costrutto più appropriato;
15. ☒ Tutti i loop e/o metodi terminano;
16. ☒ Ogni uscita da un loop è appropriamente gestita;
17. ☒ IF statements innestati possono essere convertiti in SWITCH statement?
18. ☒ Ogni metodo, classe e file è appropriamente commentato;
19. ☒ Per ogni riferimento ad array si usano indici all'interno dei limiti corretti;
20. ☒ Per ogni riferimento ad array ci si assicura che questo non sia null;

TEST COPERTURA JUNIT

La copertura è suddivisa in tre tecniche usate: Istruzioni, branch, decisioni e MCDC; ho creato una cartella "test" contenente le classi JUnit suddivise per tecnica di copertura utilizzata in esse.

Full Test

Test molto semplice che simula solo il riempimento del magazzino, la copertura risulta essere:

Name	Statement	Branch	Loop	Term
Magazzino_JUnit	60,0 %	60,0 %	33,3 %	62,5 %
Magazzino	60,0 %	60,0 %	33,3 %	62,5 %
Magazzino	100,0 %	–	–	–
insert	40,0 %	50,0 %	–	50,0 %
isFull	50,0 %	50,0 %	33,3 %	75,0 %
isFull	100,0 %	100,0 %	–	100,0 %

Copertura delle Istruzioni

Per coprire ogni riga di istruzioni del metodo "insert(productIndex, addQuantity)" sono stati creati i seguenti casi di test:

`istruzioni riga 12:` `productIndex=-1` `addQuantity=5` `[TEST CASE 1]`

`istruzioni riga 14:` `productIndex=0` `addQuantity=0` `[TEST CASE 2]`

`istruzioni riga 16:` *non testabile tramite singola esecuzione del metodo*

`istruzioni riga 19-20:` `productIndex=0` `addQuantity=5` `[TEST CASE 3]`

Implementando poi un ciclo for per riempire completamente un prodotto e poi testare la riga 16.

Per coprire il metodo "isFull(productIndex)" sono stati creati i casi di test:

`istruzioni riga 26:` `productIndex=0`

`istruzioni riga 29:` *non testabile tramite singola esecuzione del metodo*

Implementando poi un ciclo for per riempire completamente un prodotto e testare la riga 29.

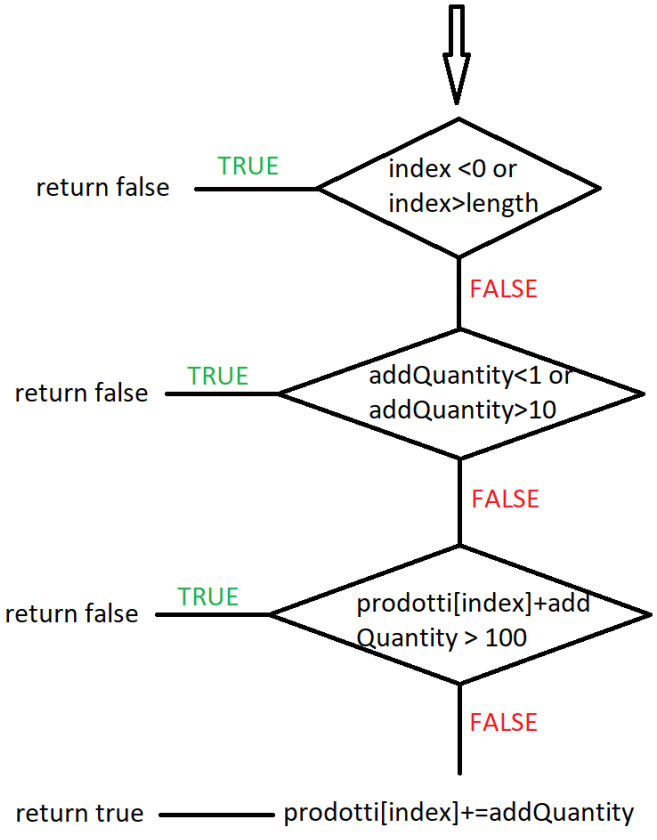
Per coprire infine il metodo "isFull()" è stato sufficiente riempire completamente il magazzino tramite un doppio ciclo for e testare il metodo prima e dopo.

Name	Statement	Branch	Loop	Term
Magazzino_JUnit	100,0 %	100,0 %	66,7 %	87,5 %
Magazzino	100,0 %	100,0 %	66,7 %	87,5 %
Magazzino	100,0 %	–	–	–
insert	100,0 %	100,0 %	–	80,0 %
isFull	100,0 %	100,0 %	66,7 %	100,0 %
isFull	100,0 %	100,0 %	–	100,0 %

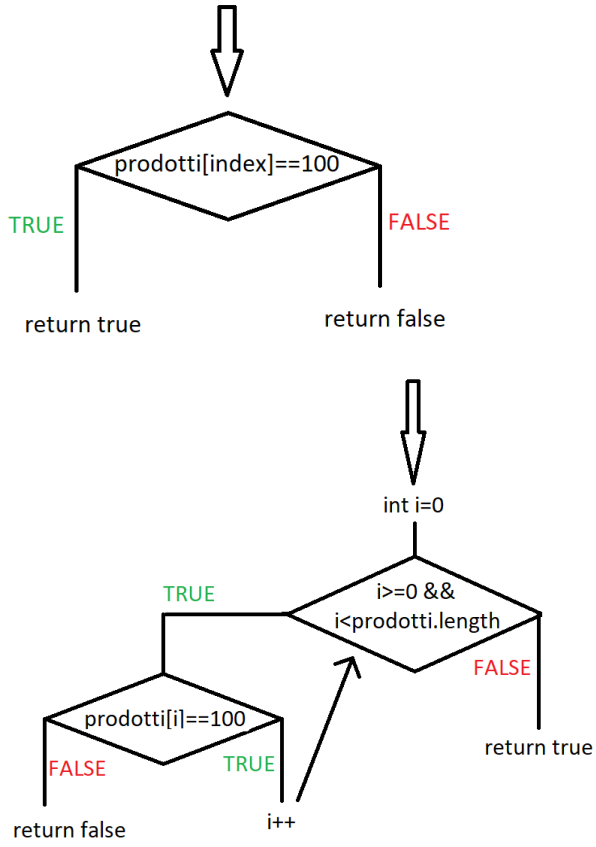
Copertura dei Branch

Per tutti e tre i metodi sono stati sufficienti i test eseguiti per la copertura delle istruzioni.

`Metodo insert(index, addQuantity)`



`Metodi isFull(int) e isFull()`



Copertura delle Decisioni

Le decisioni presenti nel metodo “insert(productIndex, addQuantity)” sono:

Decisione `if(!(productIndex>=0 && productIndex<prodotti.length)):`

VERA: [test case 1]
FALSA: [test case 3]

Decisione `if(!(addQuantity>=1 && addQuantity<=10)):`

VERA: [test case 2]
FALSA: [test case 3]

Decisione `if(prodotti[productIndex]+addQuantity > 100):`

FALSA: tutti i casi di test prima del riempimento di [0]
VERA: [test case 3] dopo il riempimento di [0]

Le decisioni presenti nel metodo “isFull(productIndex)” sono:

Decisione `if(prodotti[productIndex] == 100):`

FALSA: esecuzione prima del riempimento di [0]
VERA: productIndex = 0 dopo il riempimento di [0]

Le decisioni presenti nel metodo “isFull()” sono:

Decisione `if(p < 100):`

FALSA: esecuzione dopo del riempimento totale
VERA: esecuzione prima del riempimento totale

Per cui è stato sufficiente eseguire lo stesso test usato per le istruzioni.

Copertura MCDC

Riscrivendo le decisioni del metodo insert() come un’unica grande decisione avrei:

```
if(productIndex<0 || productIndex>4 || addQuantity <1 ||
addQuantity>10 || prodotti[index]+addQuantity>100)
```

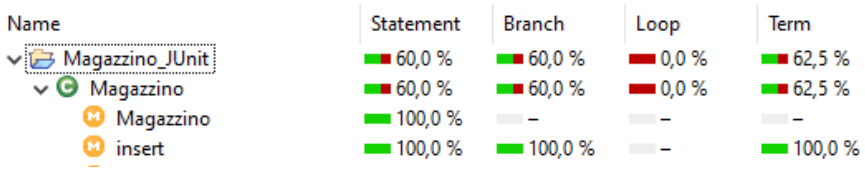
Evidenziate sono le condizioni prese in considerazione per quella decisione.

Product Index<0	Product Index>4	Add Quantity<1	Add Quantity>10	Prodotti[index]+ addQuantity>100	Decisione	TEST
T	F	F	F	F	T	Test 1
F	F	F	F	F	F	Test 3
F	T	F	F	F	T	**Test 4**
F	F	T	F	F	T	Test 2
F	F	F	T	F	T	**Test 5**
F	F	F	F	T	T	Uso for

Oltre ai tre casi di test già considerati ne ho ottenuti due nuovi:

- productIndex=10 addQuantity=5 [TEST CASE 4]
- productIndex=0 addQuantity=50 [TEST CASE 5]

Per mantenere "`Prodotti[index]+addQuantity>100`" falso basta eseguire questi test quando il magazzino è ancora vuoto. Per l'ultima riga basta riempire un prodotto completamente ed eseguire un altro inserimento valido, la condizione sarà quindi vera mentre le altre rimangono false.

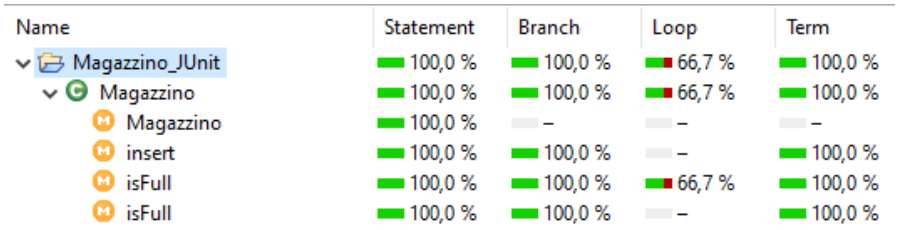


Name	Statement	Branch	Loop	Term
Magazzino_JUnit	60,0 %	60,0 %	0,0 %	62,5 %
Magazzino	60,0 %	60,0 %	0,0 %	62,5 %
Magazzino	100,0 %	–	–	–
insert	100,0 %	100,0 %	–	100,0 %

Noto che questo test permette di raggiungere il 100% di copertura "term" per insert().

RANDOOP

Tramite Randoop ho generato 150 test aventi complessivamente la seguente copertura:



Name	Statement	Branch	Loop	Term
Magazzino_JUnit	100,0 %	100,0 %	66,7 %	100,0 %
Magazzino	100,0 %	100,0 %	66,7 %	100,0 %
Magazzino	100,0 %	–	–	–
insert	100,0 %	100,0 %	–	100,0 %
isFull	100,0 %	100,0 %	66,7 %	100,0 %
isFull	100,0 %	100,0 %	–	100,0 %

TEST PARAMETRICO

Sempre usando JUnit si è creato un test parametrico che eseguisse i casi di test 1 2 e 3 automaticamente in modo sequenziale, controllando che l'output ottenuto fosse quello atteso.

OpenJML

Ho scritto i contratti per ogni metodo specificando per il metodo "insert(int, int)":

1) La **precondizione** "esiste un prodotto con meno di 100 unità":

```
(\exists int i; i>=0 && i<prodotti.length; prodotti[i]<=100);
```

2) La **postcondizione** "la media dei prodotti è minore o uguale a 100":

```
(\sum int i; i>=0 && i<prodotti.length;
 prodotti[i])/prodotti.length <= 100;
```

3) La **postcondizione** "la quantità dei prodotti diversi da productIndex è rimasta invariata":

```
ensures (\forall int i; i>=0 && i<prodotti.length &&
 i!=productIndex; prodotti[i]==\old(prodotti[i]));
```

Test JML tramite Main e modifiche al codice

1) `M.insert(0,20)` viola la precondizione del metodo insert in cui richiede valori di addQuantity positivi e minori di 10;
2) `M.insert(-1,5)` viola la precondizione del metodo insert in cui richiede valori di productIndex positivi e minori della lunghezza dell'array;
3) `M.isFull(-1)` viola la precondizione del metodo isFull(int) in cui richiede che i valori di productIndex siano positivi e minori della lunghezza dell'array;
4) `Prodotti[productIndex]+=5` nel metodo insert() viola la postcondizione in cui assicura di eseguire "prodotti[index]+=addQuantity" finchè questa somma sia sotto 100;

5) `If(true) return false` nel ciclo for del metodo isFull() viola la postcondizione solo se tutti i prodotti sono effettivamente pieni;
6) `If(false) return true` nel metodo isFull(int) viola la postcondizione solo se il prodotto in questione è pieno (afferma che se ritorna "false" è perché il prodotto ha quantità diversa da 100);
7) `Prodotti = new int[10]` nel costruttore viola la sua postcondizione.

VERIFICA CON KEY

La verifica eseguita tramite JML key ha portato ai seguenti risultati:

Target	Contract	Proof Reuse	Proof Result
isFull()	JML operation contract 0	New Proof	Open
Magazzino()	JML operation contract 0	New Proof	Closed
isFull(int)	JML operation contract 0	New Proof	Closed
insert(int, int)	JML operation contract 0	New Proof	Closed

Il contratto del metodo "isFull()" non sembra chiudersi nemmeno usando:

- `diverges true`
- `loop treatment = expand` in quanto è conosciuto (e ridotto) il numero di iterazioni che il ciclo compie.

JAVA ASSERT

Oltre ai contratti JML sono state aggiunte nel codice altre asserzioni nella forma di Java Assert controllando cose già risapute essere vere in quella posizione nel codice:

- `assert prodotti.length==5` al termine del costruttore;
- `assert prodotti[index]+addQuantity<=100` nel caso "return true" del metodo "insert()";
- `for(int p:prodotti) assert p==100` nel caso "return true" del metodo "isFull()".

Essendo corrette non ci sono stati problemi eseguendo i testi JUnit.

ASMETA

E' stata implementata la Abstract State Machine di Magazzino per il caso semplificato con solo 2 prodotti (BICICLETTA e LAMPADA) e con capacità massima di 5 unità per prodotto. Questa asm è poi stata testata con il simulatore AsmetaS e animatore AsmetaA.

Model Checking con CTL

Attraverso la libreria CTLlibrary sono state controllate le seguenti proprietà della asm:

- Esiste uno stato in cui ci sono 5 unità di prodotto ✓
- Se un prodotto è pieno rimane pieno per sempre ✓
- Le quantità dei prodotti sono sempre ≥ 0 ✓ (SAFETY)
- Prima o poi uno dei due prodotti verrà incrementato ✓ (LIVENESS)

- Il magazzino non è mai pieno ×

L'ultima proprietà è falsa e viene riportato il contro-esempio in cui si riempie il magazzino:

```
-> State: 1.1 <-

quantita(BICICLETTA) = 0
quantita(LAMPADA) = 0
agg_scelta = 1
prod_scelto = BICICLETTA

-> State: 1.2 <-

quantita(BICICLETTA) = 1
prod_scelto = LAMPADA

-> State: 1.3 <-

quantita(LAMPADA) = 1
agg_scelta = 2
prod_scelto = BICICLETTA

-> State: 1.4 <-

quantita(BICICLETTA) = 3

-> State: 1.5 <-

quantita(BICICLETTA) = 5
prod_scelto = LAMPADA

-> State: 1.6 <-

quantita(LAMAPDA) = 3

-> State: 1.7 <-

quantita(LAMPADA) = 5
```

MODEL ADVISOR

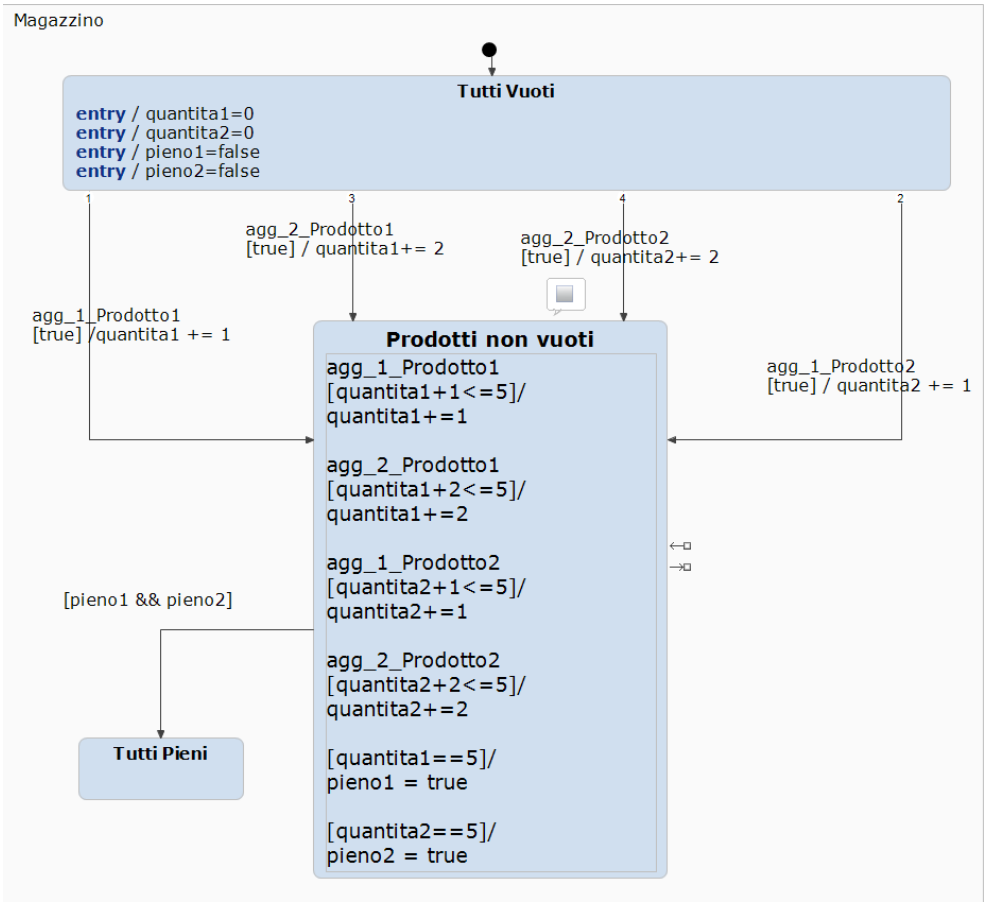
Attraverso il model advisor di Asmeta risulta che l'unica meta-proprietà NON verificata è MP2, ovvero quella per cui ogni "conditional rule" deve essere completa. Essendo però questo un errore non sintattico non è richiesta la sua verifica per il corretto funzionamento della asm.

SCENARIO AVALLA

Basandomi sul modello asm ho ricreato lo scenario di riempimento totale tramite avalla.

YAKINDU

Usando il framework di Yakindu su Eclipse ho creato il seguente modello semplificato di Magazzino:



SCENARIO YAKINDU

Proprio come in Avalla si è usato questo modello per testare lo scenario nel quale il magazzino si riempie completamente.

Input Domain Modeling & Combinatorial Testing con CTWedge

I parametri presi in considerazione per il test del metodo "insert()" sono "productIndex" e "addQuantity"; si sono quindi partizionati i loro domini di input in questo modo:

1) Valori negativi;
2) Valori accettati dal metodo: [0,5] per "productIndex" e [0,10] per "addQuantity";
3) Valori maggiori dell'intervallo accettato;

Partendo da questi 3 sotto-domini sono stati scelti i rappresentanti seguenti:

- `productIndex : [-1 .. 5]`
- `addQuantity : [-1 .. 11]`
- `returnedValue : Boolean`
- `nProductsOld : [-1 .. 101]`
- `nProductsNew : [-1 .. 101]`

Definendo poi i vincoli tra questi parametri sono stati generati attraverso il metodo PAIR-WISE un totale di 1404 casi di test.

Test	addQuantity	productIndex	returnedVal...	nProductsN...	nProductsO...
1	0	0	true	-1	-1
2	1	1	true	0	-1
3	2	2	true	1	-1
4	3	3	true	2	-1
5	4	4	true	3	-1
6	5	0	true	4	-1
7	6	1	true	5	-1
8	7	2	true	6	-1
9	8	3	true	7	-1
10	9	4	true	8	-1

Name: ACTS

Time: 46.449

1404

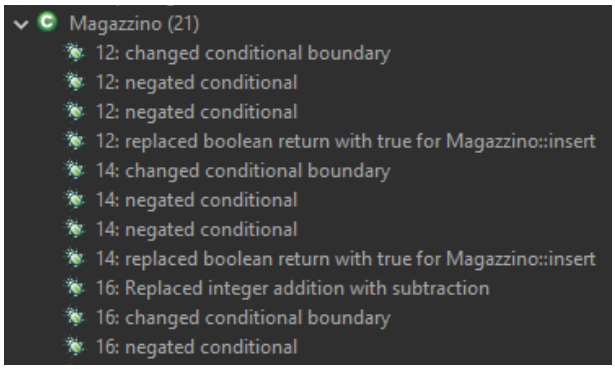
CTWedge TestSuite

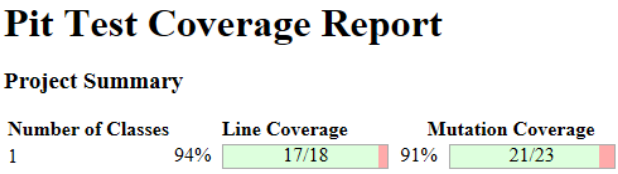
N-WISE= 2

Una delle righe è stata convertita in test JUnit con esito positivo.

MUTATION TESTING

Ho testato l'affidabilità del test basato sul coverage delle istruzioni (in quanto il più completo tra tutti) tramite il plugin **PITclipse** il quale ha generato mutazioni nella classe Magazzino:





Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
1	94%	17/18	91%	21/23

Il mio test JUnit con copertura delle istruzioni è riuscito a scoprire 21 mutazioni su un totale di 23.