Documentazione Progetto tvsw

Pellegrinelli Sean 1065868

# Requisiti

Una fabbrica (che compra semilavorati e vende prodotti finiti) è composta da tre reparti: **magazzino**, **produzione** e **risorse umane**.

* Il **magazzino** ha due scaffali, uno per i semilavorati e l’altro per i prodotti finiti. Questi si riempiono e svuotano ogni qual volta i semilavorati vengono acquistati o lavorati e quanto i prodotti finiti vengono venduti. Se lo spazio in uno dei scaffali si esaurisce il prodotto viene gettato.
* Il reparto di **produzione** è composto da 3 linee identiche che lavorano in parallelo, ogni linea svolge tre operazioni: rifinitura, colorazione e lucidatura. Con il vincolo che solo una linea alla volta può essere nella fase di lucidatura (se la lucidatura è già occupata il semilavorato rimane in attesa) e che questa fase dura il doppio delle altre due. L’interazione con questo reparto consiste nel posizionare semilavorati su una linea scarica e raccogliere prodotti finiti alla fine della lavorazione.
* Le **risorse umane** hanno il compito di valutare i dipendenti annualmente sulla base dei giorni lavorativi effettivamente svolti, le ore di straordinari fatte e se sono stati sanzionati durante l’anno oppure no. Ciascun dipendente parte da una valutazione di 50 che ogni anno può aumentare o diminuire di 20 punti o rimanere invariata. Le valutazioni minima e massima sono 0 e 100.

# Struttura dei progetti e obiettivo

Questi requisiti sono implementati nel progetto Java Maven “fabbrica”, composto dal package reparti contenente quattro classi, una per ogni reparto più un’eccezione implementata ad hoc. Ciascun reparto ha poi associato una classe (o due nel caso di risorse umane) di test.

Oltre al progetto java sono presenti due progetti per altrettante macchine a stati astratti (asm), una per il reparto di produzione e una per il magazzino.

L’obiettivo è quello di applicare diverse tecniche di testing e verifica del software ad un semplice esempio di progetto java; quindi, il focus non è il codice in sé ma l’applicazione di quante più tecniche possibile, cercando anche di sfruttare appositi tool. A ciascun reparto sono state applicate tecniche differenti (quindi le varie tecniche non sono state applicate all’intero progetto, così da rendere più evidente la differenza tra di esse).

# Continuous Integration

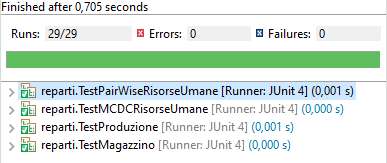
Tutti i progetti sono sulla seguente repository github: <https://github.com/PellegrinelliSean/Progetto_tvsw>. Ad ogni push sulla repository remota il progetto maven viene automaticamente compilato e testato in un apposito container, e se una delle due fasi fallisce gli sviluppatori (in questo caso solo uno) vengono avvisati. L’integrazione continua è utile (nei progetti reali almeno) per mantenere il codice condiviso sempre in buone condizioni e quindi (almeno in teoria) sempre pronto per essere distribuito.

# Code testing

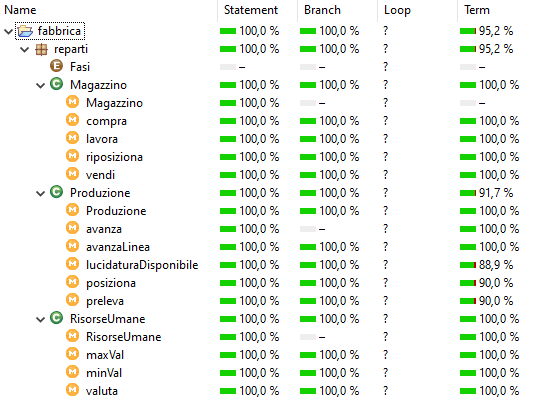
L’obiettivo del testing è trovare dei bug all’interno del codice eseguendolo ed evidenziando dei malfunzionamenti. Tutti i test del codice sono stati scritti in Junit 4 e sono stati implementati per tutte le tre classi (i tre reparti) con criteri differenti, di seguito descritti:

* Per la classe Produzione sono stati implementati dei semplici test di copertura delle istruzioni (**statement coverage**) che coincide con la **branch coverage** vista la struttura della classe. Inoltre, è presente un metodo di test che è la diretta **traduzione in Java di uno scenario Avalla** della relativa specifica asm (maggiori dettagli nella sezione di modeling). Quest’ultimo metodo non rispetta alcun criterio di copertura.
* Per la classe Magazzino sono state implementate due classi di test, una per la **copertura MCDC** e l’altra per l’**input testing** (descritto nel paragrafo di model-based testing). Per quest’ultimo sono stati sfruttati i **test parametrici** forniti da Junit 4.
* Per la classe RisorseUmane sono stati tradotti gli scenari avalla generati automaticamente da Asmeta ATGT (anche questo descritto nel paragrafo di model-based testing).

I risultati dei test sono i seguenti:



L’effettiva copertura del codice è stata misurata col tool **CodeCover**, la cui analisi è rappresentata di seguito:



Di seguito vengono riportate le tabelle realizzate per la creazione dei casi di test di RisorseUmane per la copertura MCDC:

**TABELLA A:** i < 0 || i >= valutazioni.length

|  |  |  |
| --- | --- | --- |
| i < 0 | i >= valutazioni.length | VALUTAZIONE |
| F | F | F |
| T | - | T |
| F | T | T |

**TABELLA B:** sanzione || (giorniLavoro < 200 && oreStraordinari < 10)

|  |  |  |  |
| --- | --- | --- | --- |
| sanzione | giorniLavoro < 200 | oreStraordinari < 10 | VALUTAZIONE |
| F | F | - | F |
| T | - | - | T |
| F | T | T | T |
| F | T | F | F |

**TABELLA C:** giorniLavoro > 240 || oreStraordinari > 50

|  |  |  |
| --- | --- | --- |
| giorniLavoro > 240 | oreStraordinati > 50 | VALUTAZIONE |
| F | F | F |
| T | - | T |
| F | T | T |

**TABELLA D:** valutazioni[i] > 100

|  |
| --- |
| Valutazioni[i] > 100 |
| F |
| T |

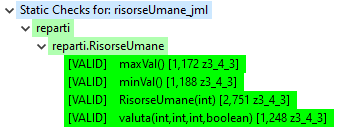
**TABELLA E:** valutazioni[i] < 0

|  |
| --- |
| Valutazioni[i] < 0 |
| F |
| T |

# Verification

## Design by Contract

Per la classe RisorseUmane sono stati scritti i contratti in JML (Java Modeling Language), ovvero pre e post condizioni per i tre metodi e per il costruttore più un’invariante. A **runtime** la violazione di questi contratti può essere evidenziata utilizzando il bytecode generato da **OpenJML** (Il tool utilizzato), ma per questo progetto si è posto il focus sulla **verifica formale** dei contratti, la cui correttezza è stata dimostrata matematicamente ed in automatico da OpenJML. Il risultato della verifica è il seguente:

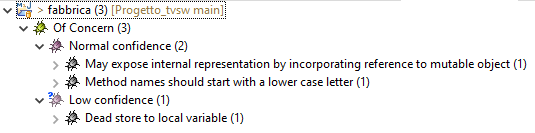


Visto che tutti i metodi sono stati correttamente validati si ha la certezza (almeno in teoria) che per qualsiasi input che rispetta le pre-condizioni fornito a ciascun metodo, l’output fornito rispetterà le post-condizioni e l’invariante.

Nota: Visto che OpenJML si appoggia su Java 1.8 e il progetto Maven è stato realizzato con una versione più recente, per la verifica è stato creato uno workspace apposito contente un progetto con la classe RisorseUmane ed i suoi contratti. Nel progetto originale i contratti sono comunque presenti ma la verifica è stata svolta sull’altra classe (identica in tutto e per tutto).

## Static Analysis

Come tool di analisi statica è stato utilizzato **SpotBugs**, il quale ha trovato tre possibili bug:



Tutti e tre i bug hanno il livello di criticità minimo e dell’esistenza del terzo il tool non è molto certo.

Il primo bug evidenzia che un campo del test parametrico di RisorseUmane potrebbe essere modificato dall’esterno della classe, questo però non dovrebbe essere un problema visto che è una classe di test e quindi usata solo per eseguire dei test (nessun altro codice dovrebbe interagire direttamente con questa classe).

Il secondo bug evidenzia semplicemente che la convenzione di nomenclatura dei metodi non è stata rispettata ed è stato velocemente risolto.

Il terzo bug si riferisce al codice di un test dove una variabile locale viene inizializzata creando un oggetto con parametri scorretti, provocando il lancio di un’eccezione. Ovviamente la variabile locale non è così utilizzabile e quindi non viene utilizzata, viene di seguito assegnata ad un nuovo oggetto (corretto). Quindi il problema non sussiste. la situazione così creata è desiderata per il caso di test.

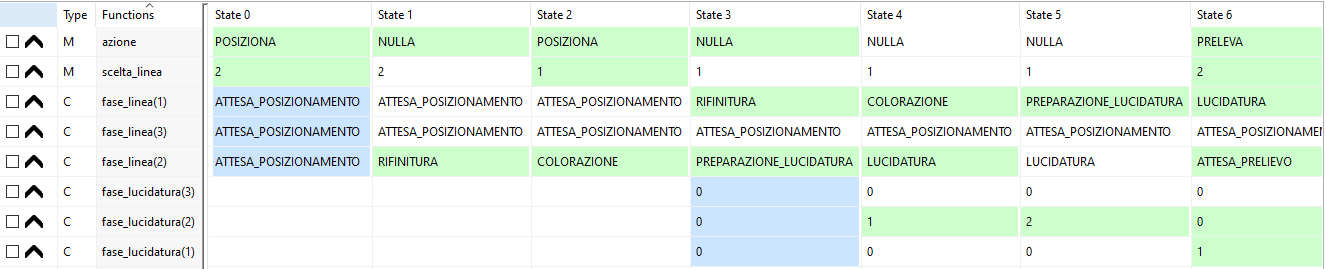
## Code Inspection e refactoring

Consiste nell’analizzare “manualmente” il codice sorgente per trovare cattive pratiche di scrittura del codice, l’obiettivo non è infatti quello di trovare dei bug, ma più che altro migliorare il codice rendendolo più leggibile o evidenziando e correggendo eventuali “bad practice” (**code refactoring**). Solitamente questa operazione non viene svolta da colui che ha scritto il codice (è molto più difficile che chi ha scritto il codice si accorga delle proprie “bad practice”). Durante la scrittura del codice si è cercato di svolgere code inspection e code refactoring il più possibile, anche sfruttando SpotBugs (quando il progetto era ultimato) come scritto sopra.

# Modeling

Sono stati scritti due modelli **Asmeta**, uno per Produzione e l’altro per Magazzino. Di seguito viene descritto la asm realizzata per il reparto produzione.

L’obiettivo è realizzare un fedele modello della classe Produzione.java (ovvero produzione.asm) così da poter scrivere delle **proprietà della logica temporale** e verificare se il modello le rispetta. Se così è, si può supporre che anche il codice rispetti queste proprietà. Per questa macchina sono state scritte e verificate quattro proprietà della logica temporale (3 CTL e 1 LTL) e un’**invariante** (tradotta in automatico in una regola CTL). Ad esempio l’invariante rappresenta la proprietà: “non è possibile che due o più linee svolgano contemporaneamente la fase di lucidatura”. Inoltre, si può animare la macchina a stati astratta per avere un’idea del funzionamento del codice con degli esempi più “visuali”, come nel seguente esempio.



Inoltre, è stato scritto uno scenario **Avalla**, che in pratica corrisponde ad un test per la specifica Asmeta. Come per i test JUnit, questo scenario esegue la asm e ne confronta il comportamento reale rispetto a quello atteso. Lo scenario è stato poi tradotto in un test JUnit 4 come già detto.

La asm del reparto magazzino è molto più semplice e permette quindi di sfruttare **ATGT** (descritto nel prossimo paragrafo) e il Model Advisor, entrambi non utilizzati per la prima macchina perché troppo complessa (infatti questi due tool sono un po’ fragili e richiedono delle restrizioni della asm). Il **Model Advisor** è un tool di analisi statica che analizza 7 proprietà che un buon modello asm dovrebbe avere. SI riporta come esempio la proprietà MP5: “For every domain element e, there exists a location which has value e”. Tutte le 7 proprietà sono rispettate eccetto MP2: “Every conditional rule must be complete”, che però è un po’ forte e non viene rispettata neanche se per ogni “if” è presente il corrispettivo “else”, quindi non dovrebbe essere un grave problema se non viene rispettata.

# Model-based testing

Questa tecnica consiste nel creare un modello del codice (o di un suo aspetto) per ricavare in automatico (almeno in parte) dei casi di test.

## MBT - ATGT

La specifica asm di Magazzino è stata utilizzata per creare in automatico degli scenari avalla (tramite il tool **ATGT**) che rispettano il criterio di copertura degli assegnamenti nelle rule. Questi scenari avalla sono stati rifiniti (aggiunti dei check finali) e tradotti manualmente in test Junit per la classe Magazzino. Questi test rispettano il criterio di copertura delle istruzioni (come ci si aspetta) ma anche dei branch ed MCDC visto la struttura molto semplice della classe.

## Input domain modeling e combinatoria testing

Per il metodo “valuta” della classe RisorseUmane è stato creato un modello degli input. Questo metodo ha quattro parametri: tre interi ed un booleano. Applicando il **functional-based approach** sono stati partizionati i parametri nel seguente modo:

* giorniLavoro è stato diviso in 4 sottoinsiemi:
  + giorniLavoro < 0
  + 0 <= giorniLavoro < 200
  + 200 <= giorniLavoro <= 240
  + giorniLavoro > 240
* oreStraordinari è stato diviso in 4 sottoinsiemi:
  + oreStraordinari < 0
  + 0 <= oreStraordinari < 10
  + 10 <= oreStraordinari <= 50
  + oreStraordinari > 50
* i (numero dipendente) è stato diviso in 3 sottoinsiemi (considerndo il caso di 3 dipendenti):
  + i < 0
  + 0 <= i < 3
  + i >= 3
* sanzone è booleano, quindi è stato diviso in true e false.

La test suite è stata scelta in modo da rispettare il criterio **pair-wise** utilizzando la tecnica **IPO**. Il risultato è il seguente:

|  |  |  |  |
| --- | --- | --- | --- |
| giorniLavoro | oreStraordinari | i | sanzione |
| -1 | -1 | -1 | T |
| -1 | 5 | 1 | F |
| -1 | 25 | 10 | T |
| -1 | 100 | -1 | F |
| 100 | -1 | 1 | F |
| 100 | 5 | 10 | T |
| 100 | 25 | -1 | F |
| 100 | 100 | -1 | T |
| 220 | -1 | 10 | T |
| 220 | 5 | -1 | F |
| 220 | 25 | 1 | T |
| 220 | 100 | -1 | F |
| 300 | -1 | -1 | F |
| 300 | 5 | 1 | T |
| 300 | 25 | 10 | F |
| 300 | 100 | -1 | T |