
# Progetti di Informatica III A

Università degli Studi di Bergamo

A.A. 2019/2020

Piffari Michele - 1040658

March 1, 2020

# Contents

# List of Figures

# List of Tables

# Part I

# Primo progettino

# 1

# Cyclone

# Part II

# Secondo progettino

# 2

# C++

S

# Part III

# Terzo progettino

# 3

# Energy drink vending machine con Scala

## 3.1 Descrizione del progetto

Il progetto scritto in Scala prevede di andare ad affiancare, alla macchinetta del caffè gestita in ASMETA, un distributore automatico di bevande energetiche. In particolar modo si è progettato un sistema con queste specifiche:

- Ogni distributore automatico può essere impostato per funzionare in una lingua piuttosto che in un'altro: nel codice è gestitata solamente la possibilità di introdurre distirbutori automatici in lingua italiana e in lingua inglese.
- Questi distributori possono gestire le seguenti bevande energetiche (*energy drink*):
  - RedBull
  - Monster
  - Gatorade
  - Italian

  Ognuno di questi prodotti sarà caratterizzato dai seguenti campi descrittivi:
  - prezzo
  - volume (espresso in cl)
  - data di scadenza
  - insieme di tags, che permettono di esprimere le caratteristiche salienti di ognuno degli energy drink

  Ogni distributore andrà ad offrire le seguenti funzionalità:
- Acquisto dei prodotti disponibili, regalando i prodotti scaduti: in particolare la macchinetta sarà in grado di fornire resto esatto al cliento (oppure tutta la somma di denaro inserita nel caso di prodotto scaduto).
- Mostrare l'elenco dei prodotti disponibili all'interno del distirbutore (identificato tramite ID), unitamente al numero di pezzi disponibili, come si vede in figura 3.1.

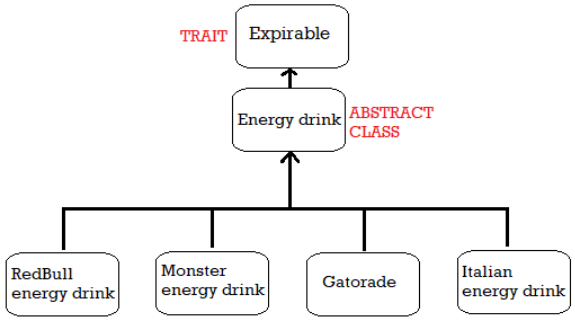
```
Available drinks for machine with id 100:

        - 3 RED BULL
        - 1 MONSTER
        - 1 GATORADE
        - 1 ITALIAN
```

Figure 3.1: Prodotti disponibili

- Possibilità di cercare un prodotto tramite tag, per poter così trovare l'energy drink più adatto ad ogni evenienza
- Aggiunta di energy drink all'interno del distributore: nello specifico, l'aggiunta di un nuovo prodotto, avviene all'interno di una struttra definita come un array di Queue (ovvero una matrice), che non fa altro che andare a riprodurre la fisionomia di un distirbutore reale.

## 3.2 Gerarchia delle classi

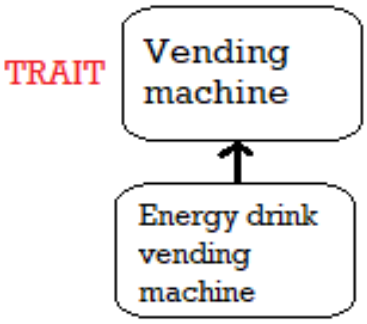
Nella applicazione realizzata sono stati realizzata due gerarchie facendo uso dei *trait*: i trait in Scala corrispondono alle interfaccie in Java, ovvero permettono di definire la firma di ogni classe che ne implementa la struttura. Nel nostro caso abbiamo due strutture gerarchiche, gestite tramite traits, mostrate con un grafo ad "albero" nell'immagine seguente (figura 3.2).




(a) *Energy drink*

```
trait Expirable {
  def isExpired(): Boolean
}
```

(b) *Expirable trait*




(c) *Vending machine*

```
trait VendingMachine {
  def addProduct(product: EnergyDrink)
  def searchByTag(tag: String) : List[String]
  def buy(column: Int, money: Double): Double
  def show
}
```

(d) *Vending machine trait*

Figure 3.2: Classi e trait

## 3.3 Filter

Per quanto riguarda la funzionalità di ricerca delle bevande in base ai tag, è stato utile la funzionalità *filter*.

Il comando di filter è utile nel momento in cui si vogliono filtrare gli elemnti di una certa collezione (lista, array, vettore etc), andando a creare una nuova collezione contenente solamente gli elementi che rispettano il criterio di filtraggio definito in maniera custom.

Nella nostra applicazione siamo andati a filtrare l'elenco dei tag di ogni prodotto: il filtraggio è stato pensato per mantenere solamente i tag contenenti al loro interno il tag cercato dall'utente tramite tastierino di input.

```
def searchByTag(tag: String) : List[String] = {
  val columns = Vector(RED_BULL,MONSTER,GATORADE,ITALIAN)

  def checkTag(product: EnergyDrink) : Boolean = {
    if(language == "ITA") {
      if(product.tagITA.filter(t => t.contains(tag)).size > 0) {
        return true
      } else {
        return false
      }
    } else if(language == "ENG") {
      if(product.tagENG.filter(t => t.contains(tag)).size > 0) {
        return true
      } else {
        return false
      }
    } else {
      return false
    }
  }
  ...
```

Figure 3.3: Utilizzo comando *filter* nella nostra applicazione

Come si vede in figura 3.3, in base alla tipologia di lingua con cui il distributore è stato impostato, si va a cercare nei rispettivi tag, solamente quelli che contengono la parola chiave cercata: in particolare, se il risultato dell'operazione di filter è un vettore con una lunghezza maggiore di 0, significa che il prodotto in esame soddisfa il tag cercato, e quindi può essere mostrato all'utente. Questa operazione di filtraggio viene poi ripetuta per ogni tipologia di prodotto disponibile, ovvero per ogni colonna.

Una volta effettuato il filtraggio siamo andati a considerare solamente i prodotti il cui risultato dall'operazione di filter non fosse un vettore nullo: questo corrisponde infati al considerare solamente i prodotti che contengono il tag cercato, notificando poi all'utente la loro tipologia, tramite una stampa a display.

## 3.4 Match - sealed

### 3.4.1 Match

Il *pattern matchin* rappresenta una struttura per verificare il valore aassunto da una variabile, tramite un pattern: si tratta in sostanza di una versione leggermente più potente del construtto *switch* di Java. Nella nostra applicazione si è pensato di andare ad utilizzare il *pattern matching* in due situazioni:

- *Pattern guard*: nell'andare a definire se un prodotto è considerabile come normo, ipo o iper calorico, si è fatto uso di un pattern guard, sfruttando anche la potenzialità aggiuntiva di poter aggiungere una condizione dopo il pattern, tramite la dicitura *if<boolean expression>*, che permette di rendere più specifico ogni singolo case.

- *Matching on classes*: per la funzionalità di aggiunta di nuovi prodotti all'interno del distributore automatico, si era pensato di utilizzare un *pattern match* per poter distinguere la tipologia specifica del prodotto aggiunto. Alla fine la scelta è però ricaduta sull'utilizzo del comando *isInstanceOf*.

### 3.4.2 Sealed

Classi e traits possono essere segnati come *sealed*: questo significa che tutti i sotto tipi devono essere dichiarati all'interno dello stesso file, garantendo che tutti i sottotipi siano conosciuti e noti.

```
sealed trait Calorico

final case class IpoCalorico() extends Calorico
final case class NormoCalorico() extends Calorico
final case class IperCalorico() extends Calorico
```

Figure 3.4: Sealed

# Part IV

# Quarto progettino

# 4

# Coffe Machine con ASM

## 4.1 Descrizione del progetto

Per quanto riguarda la parte di ASM *ASM* si è deciso di riprendere un esempio visto in classe (e durante le pause), ovvero quello relativo alla ***Coffee machine***. Il distributore modellato può preparare diversi tipi di bevande (caffè, cappuccino etc), ognuna preparata con diverse quantità di ingredienti (acqua, caffè, latte etc) i quali vengono consumati e devono essere reintegrati dal manutentore.

Il distributore accetta pagamenti solamente in moneta tramite l'inserimento di denaro nell'apposita fessura.

Il distributore è in grado di fornire il resto (anche se non sempre in modo esatto).

Quando tutte le bevande sono esaurite, il distributore va fuori servizio, in attesa che gli ingredienti vengano aggiunti dal manutentore, il quale può inoltre prelevare o inserire monete dal distributore, sempr etenendo conto del vincolo di capacità del vano porta monete.

## 4.2 Macchina a stati

La ASM è basata su una sottostante macchina a stati finiti, mostrata in figura, che definisce i principali stati e transizioni del distributore. La ASM permette di estendere questa FSM introducendo un concetto aumentato di "stato", che comprende anche funzioni dinamiche, modificando le quali si possono memorizzare informazioni aggiuntive.

In particolare, è stato possibile memorizzare informazioni su:

- Quantità di ingredienti residui
- Monete possedute dal distributore
- Credito dell'utente attuale

## 4.3 Standard Library

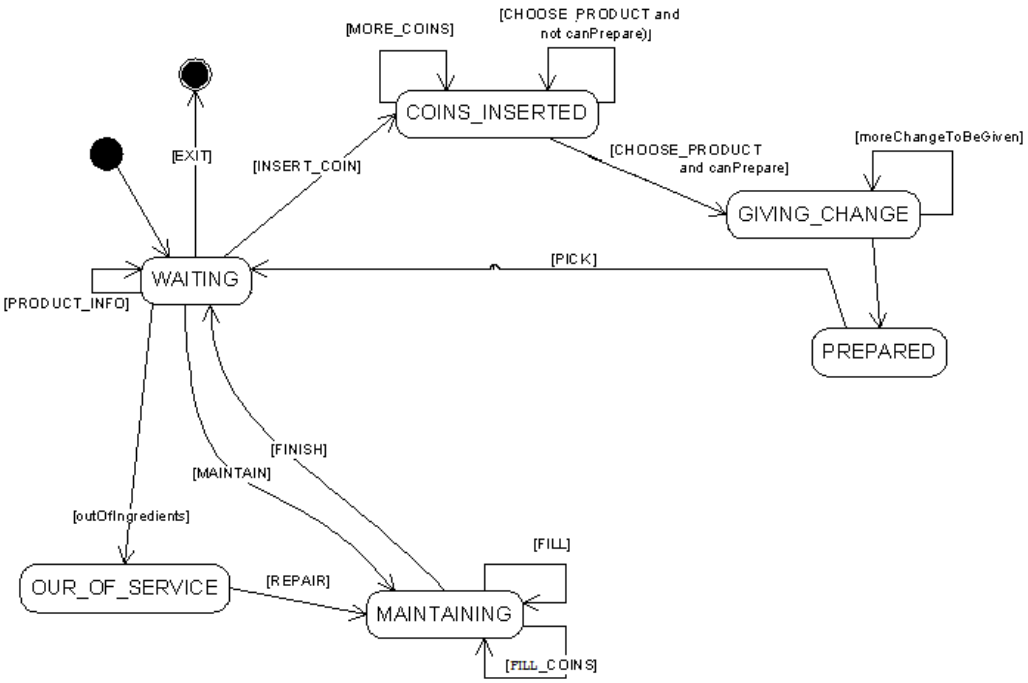



Figure 4.1: Macchina a stati