



Progetti di Informatica III A

Università degli Studi di Bergamo
A.A. 2019/2020

PIFFARI MICHELE - 1040658

March 13, 2020

Contents

I	Primo progettino	1
1	Cyclone	3
1.1	Introduzione	3
1.2	Descrizione del progetto	3
1.3	Costrutti Cyclone	4
1.3.1	Puntatori @fat e @thin	4
1.3.2	Puntatori @nullable e @notnull	4
1.3.3	Regioni	5
II	Secondo progettino	7
2	C++	9
2.1	Descrizione del progetto	9
2.2	Gerarchia delle classi	9
2.3	Multiple inheritance	10
2.4	Diamond inheritance	10
2.5	Virtual method	11
2.6	Overloading particolari	11
2.6.1	Overloading di cout«	11
2.6.2	Overloading di ()	11
2.7	Templates	12
2.8	Standard Template Library	12
2.8.1	STL - Iterator	12
2.8.2	STL - Algorithm	13
III	Terzo progettino	15
3	Energy drink vending machine con Scala	17
3.1	Descrizione del progetto	17
3.2	Gerarchia delle classi	18
3.3	Filter	19
3.4	Match - sealed	19
3.4.1	Match	19
3.4.2	Sealed	20
IV	Quarto progettino	21
4	Coffe Machine con ASM	23
4.1	Descrizione del progetto	23
4.2	Macchina a stati	23
4.3	Eventi	23

4.4	Domini	24
4.5	Controlled - static - monitored functions	25
4.5.1	Controlled	25
4.5.2	Static	25
4.5.3	Monitored	25
4.6	Inizializzazione	26
4.7	Event management rules e main rule	26
4.8	Transition rule	27
4.9	Simulazione	27

List of Figures

1.1	@fat pointer come Struct	4
1.2	Gestione pointer nullable	5
2.1	UML class diagram	9
2.2	Esempio di eredità multipla della classe <i>Matrix</i>	10
2.3	Diamond problem	10
2.4	Virtual Inerithance nelle classe <i>Addable</i> e <i>Subtractable</i>	10
2.5	Virtual destructor della classe base <i>Number</i>	11
2.6	Ridefinizione dell'operatore di stream	11
2.7	Ridefinizione degli operatori nella classe <i>Matrix</i>	12
2.8	Esempio di utilizzo della ridefinizione degli operatori	12
2.9	STL - Iterator	13
2.10	STL - Algorithm	13
3.1	Prodotti disponibili	17
3.2	Classi e trait	18
3.3	Utilizzo comando <i>filter</i> nella nostra applicazione	19
3.4	Sealed	20
4.1	Macchina a stati	24
4.2	Action che implicano un possibile cambio di stato	24
4.3	Domini enumerativi	24
4.4	Funzioni non monitored	25
4.5	Static functions	25
4.6	Monitored functions	25
4.7	Inizializzazione	26
4.8	Management rules	27

List of Tables

Part I

Primo progettino

1

Cyclone

1.1 Introduzione

Il progetto è stato realizzato e testato in linguaggio C, per poi essere portato in Cyclone.

Cyclone è un dialetto safe del C che permette di prevenire diversi tipi di errori e problemi di sicurezza molto comuni in C come buffer overflow, stringhe non terminate e dangling pointers. Per ottenere questi risultati si è andato ad aggiungere il garbage collector, che solleva il programmatore dal dover esplicitamente deallocare la memoria con le chiamate `free()`, riducendo la possibilità di incorrere in dangling pointer o di memoria non deallocata al termine del suo utilizzo, liberando memoria non più referenziata da altri puntatori.

Altra caratteristica importante di Cyclone sono i qualificatori dei puntatori che meglio specificano i possibili valori assunti dai puntatori e aggiungono controlli sull'utilizzo degli stessi. In questo modo Cyclone permette di eseguire in sicurezza operazioni che riguardano i puntatori come aritmetica sui puntatori e gestione di stringhe.

Il porting è stato effettuato manualmente ed è consistito principalmente nella ridefinizione dei tipi puntatore, che rappresenta il cavallo di battaglia di Cyclone nell'assicurare la type safety, per comprendere così al meglio le funzionalità offerte da Cyclone.

1.2 Descrizione del progetto

L'applicazione scritta in C e successivamente portata in Cyclone, consiste in un piccolo programmino in grado di leggere dati da un file, parsarli e poi di fornire funzionalità di ricerca sui dati contenuti al suo interno. Nello specifico si è pensato di inserire questa applicazione nell'area break dell'università: infatti, in concomitanza con la macchinetta del caffè gestita in *ASMETA* e il distributore di energy drink prodotto in *Scala*, ho pensato di introdurre un sistema per gestire cartoline. Nello specifico è stata realizzata un'interfaccia in grado di funzionare in questo modo:

- L'applicazione legge i dati da un file *.txt* in cui sono contenute una serie di cartoline descritte da
 - Mittente
 - Destinatario
 - Località da cui è stata spedita
- Ognuna delle informazioni che caratterizza ogni singola è divisa per mezzo di un carattere delimitatore (`|`) che permette alla funzione di *tokenize* di andare ad assegnare le corrette informazioni ad ogni singola cartolina
- Una volta letto il file in ingresso, che potrebbe rappresentare l'insieme di tutte le cartoline relativo ad un account su una specifica piattaforma online per la gestione delle cartoline, l'interfaccia permette di eseguire una ricerca:
 - *BY SENDER*

- *BY RECEIVER*
- *BY PLACE*

stampando poi le cartoline trovate (qualora ce ne fossero).

1.3 Costrutti Cyclone

In questa piccola applicazione sono stati due i costrutti principali che Cyclone offre e che sono stati utilizzati:

- **Puntatori:** Cyclone permette l'utilizzo di normali puntatori con le seguenti modifiche rispetto a C
 - Controlla se il puntatore è nullo ad ogni de-reference dello stesso (previene Segmentation Fault)
 - Cast vietato da int a puntatore (previene Out of Bounds)
 - Aritmetica dei puntatori vietata (previene Buffer Overflow, Overrun e Out of Bounds)

Ogni puntatore ha inoltre una serie di annotazioni che specificano come deve essere trattato; ogni annotazione inizia con un carattere @.

Generalmente queste annotazioni sono ortogonali tra di loro, ovvero possono essere combinate (tranne alcune situazioni particolari) in tutti i modi.

- **Regioni:** vedi sezione 1.3.3

1.3.1 Puntatori @fat e @thin

I puntatori sono di default @thin, ovvero non sono in grado di controllare dinamicamente il rispetto dei limiti (bounds) dell'array. I puntatori @fat effettuano invece tale controllo ogni volta che viene utilizzata l'aritmetica dei puntatori. I puntatori fat possono essere definiti in modo abbreviato con il carattere ?. I puntatori fat possono essere pensati come una struttura, per cui i fat pointer permettono l'aritmetica sia in avanti sia all'indietro, con la garanzia di non eccedere i limiti dell'array, come si vede in figura 1.1.

```
struct _tagged_arr {
    char *base; // pointer to first element
    char *curr; // current position of the pointer
    char *last; // pointer to last element
};
```

Figure 1.1: @fat pointer come Struct

Questa tipologia di puntatori, è stata usata diffusamente all'interno del codice Cyclone, soprattutto nella funzione di *tokenize* dove si è ritenuto opportuno sfruttare il fatto che i *fat* pointer contengano l'informazione sulla lunghezza, accessibile tramite il comando *numelts*.

1.3.2 Puntatori @nullable e @nonnull

I puntatori sono di default @nullable, ovvero possono assumere valore NULL. Tali puntatori si possono definire esplicitamente con *@nullable. I puntatori fat possono essere solo @nullable: un puntatore @fat@nonnull può essere nullo (il compilatore a volte ignora il @nonnull, a volte emette degli errori, ma comunque non forza il puntatore a essere non nullo)

Questi sono stati utilizzati nella parte di apertura del file *.txt*: come si vede in figura 1.2, non si ha fatto uso di puntatore @nonnull per garantire la gestione del caso in cui non si riesca ad aprire il file di testo, tramite un print di errore.

I puntatori @nonnull non possono invece essere nulli, ovvero non è possibile assegnare loro il valore NULL. Essi possono essere definiti in modo abbreviato con il carattere @. I puntatori non nulli sono

```
FILE* filePointer = fopen(FILE_PATH, "r");
if(filePointer == NULL) {
    printf("Error opening file %s\n", FILE_PATH);
    return postcards;
} else {
    printf("FILE OPENED %s\n", FILE_PATH);
}
```

Figure 1.2: Gestione pointer nullable

sicuramente i più utilizzati, sia perché non introducono l’overhead del controllo di non-nullità (che viene garantita a compile-time), sia perché nella quasi totalità dei casi un puntatore è utilizzato per operare sull’oggetto puntato e non per verificare se tale oggetto esiste, quindi si dà per scontato che esso esista.

Questa tipologia di puntatore è stata largamente utilizzata nper quanto riguarda le stringhe, relative alla struttura *postcard*.

1.3.3 Regioni

Per evitare dangling pointers, Cyclone impone che ogni puntatore dichiari in quale area della memoria punti. L’area (region) in cui punta può essere un particolare record di attivazione sullo stack, lo heap o una regione di stack allocata dinamicamente. Ogni puntatore può puntare solo a regioni che hanno una vita uguale o più lunga di quella della regione dichiarata; ad esempio, un puntatore allo heap può puntare solo allo heap, un puntatore al record di attivazione di una funzione può puntare a quel record, ai record dei chiamanti della funzione o allo heap, ma non può puntare a record di funzioni chiamate

Per indicare esplicitamente una regione, si deve annotare il puntatore con `@region('r)` o `@effect('r)` o semplicemente `'r`, dove `r` è il nome di un’opportuna regione o un semplice segnaposto. `'H` rappresenta lo heap.

Part II

Secondo progettino

2

C++

2.1 Descrizione del progetto

Per quanto riguarda il progetto realizzato in cpp si è pensato, piuttosto che realizzare un applicativo "funzionale" (approccio seguito per gli altri 3 progetti), di andare a realizzare, come oggetto di questo elaborato, una libreria per il calcolo numerico, che consiste in due moduli principali:

- Matrici
- Applicativo dimostrativo

In particolare sono state sviluppate alcune funzionalità algebriche base come *somma* e *sottrazione* elemento per elemento di una matrice: l'espressività del C++ ha permesso di rendere questa libreria del tutto generica, permettendo quindi di realizzare matrici composte da elementi di qualsiasi tipo, tramite l'utilizzo dei *templates generici*.

2.2 Gerarchia delle classi

Come si vede nell'*UML Class Diagram* in figura 2.1, la libreria presenta una classe base principale, *Number*, che rappresenta un numero il quale può essere un numero di tipo *Addable* o *Subtractable*, che offrono rispettivamente un metodo e un operatore per realizzare la somma e la sottrazione.

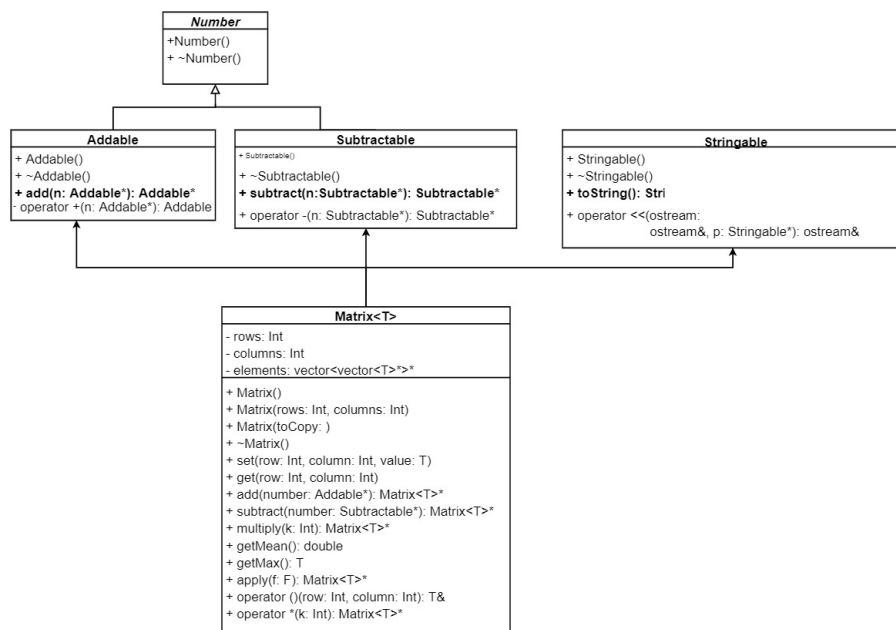


Figure 2.1: UML class diagram

2.3 Multiple inheritance

Nella gerarchia delle classi esposta sopra, è stato necessario utilizzare l'ereditarietà multipla nei due diversi contesti tipici:

- implementazione di interfacce
- ereditarietà multipla *classica*

Il C++ non fa distinzioni sostanziali tra queste due tipologie, ma la differenza concettuale è notevole, tanto che altri linguaggi (come Java) permettono il primo tipo di ereditarietà multipla e non il secondo.

La prima tipologia consiste nel derivare una classe da al più una classe base “non pure virtual”. Le altre classi base devono essere l'equivalente delle interface Java, ovvero devono essere classi astratte (*Abstract Base Classes* - ABCs) in cui tutte le member functions (o quasi) sono pure virtual e in cui tutte le member variables sono costanti e pubbliche.

Questa tipologia di eredità è stata utilizzata per derivare da *Stringable*.

La seconda tipologia di eredità è stata invece utilizzata per andare a modellizzare il rapporto tra una matrice e quelle che sono le strutture relative a *Number*, come si vede nella figura 2.2.

```
/**
 * Matrix composed by generics elements
 */
template<class T> class Matrix: public Addable, public Subtractable, public Stringable {
```

Figure 2.2: Esempio di eredità multipla della classe *Matrix*

2.4 Diamond inheritance

Un aspetto problematico dell'ereditarietà multipla (specialmente per il secondo tipo) è la possibilità di generare una gerarchia a diamante.

Esso si verifica quando, come si vede nella figura 2.3 una generica classe D eredita da due classi B e C che hanno un antenato in comune A. In questo caso c'è un'ambiguità su quale “versione” dell'antenato deve essere ereditata da D: quella di B o quella di C?

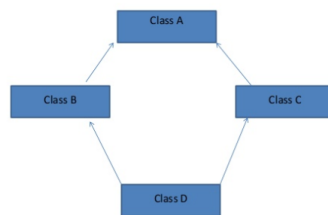


Figure 2.3: Diamond problem

Un modo di risolvere il problema in C++ è dichiarare, nelle classi B e C, un'eredità *virtuale* da A, specificandolo direttamente nella dichiarazione, come fatto nelle classi *Addable* e *Subtractable* (figura 2.4).

```
class Subtractable: virtual public Number
class Addable: virtual public Number
```

Figure 2.4: Virtual Inerithance nelle classe *Addable* e *Subtractable*

L'ereditarietà virtuale introduce comunque un altro leggero problema: poiché esiste un'unica classe base, le classi derivate direttamente da essa non possono inizializzare l'oggetto della classe base, perché si avrebbe un conflitto tra queste inizializzazioni. Il compito di costruire l'istanza della classe base

è quindi delegato alla classe most-derived, ovvero a quella che sta alla “convergenza” dei rami del diamante. Ciò che è più contro intuitivo è che anche eventuali classi derivate da quest’ultima (anche con ereditarietà singola) devono inizializzare direttamente la virtual base class; questo può rendere poco chiaro il codice e richiedere alle classi derivate di conoscere alcuni dettagli della virtual base class che altrimenti potrebbero ignorare.

2.5 Virtual method

In C++, una sottoclasse può sempre ridefinire i metodi della sopraclasse, ma il dynamic binding non avviene se non si usa la parola chiave `virtual` (a differenza di Java dove avviene sempre).

Pertanto, ogni metodo che viene (o potrebbe essere) ridefinito in una sottoclasse va dichiarato come virtuale. Un caso particolare è il distruttore, che se non fosse virtuale non permetterebbe la distruzione di tutti gli oggetti appartenenti alle sottoclassi: per questo che, nella classe base *Number* il distruttore è stato dichiarato di tipo `virtual` come si vede in figura 2.5

```
#ifndef NUMBER_H_
#define NUMBER_H_

/**
 * Base class for this library
 */
class Number{
public:
    Number(); // Default constructor
    virtual ~Number(); // Default destructor
};

#endif /*NUMBER_H_*/
```

Figure 2.5: Virtual destructor della classe base *Number*

L'utilizzo di metodo virtuale ne permette anche l'applicazione e l'utilizzo come interfacce Java, tramite i metodi *virtual pure*: in particolare, questi metodi virtuali, possono essere usati proprio come gli altri, con la differenza che le classi contenenti metodi virtuali puri non possono però essere istanziate.

2.6 Overloading particolari

Nella nostra libreria, su consigli di altri studenti, siamo andati a ridefinire alcuni operatori, per renderli più "personalizzati".

2.6.1 Overloading di `cout`

Tra gli operatori che risulta utili sovraccaricare c'è "`<<`", il quale è utilizzato nella libreria standard per inviare dati agli oggetti *ostream* (flussi in output): è stato quindi possibile personalizzare, in maniera facile e diffusa, la tipologia di log aggiungendo, come si vede in figura 2.6,

```
ostream& operator<<(ostream &ostream, Stringable* p){
    auto now = std::chrono::system_clock::now();
    std::time_t now_time = std::chrono::system_clock::to_time_t(now);

    return ostream<<"[Matrix library log] - "<<ctime(&now_time)<<p->toString();
}
```

Figure 2.6: Ridefinizione dell'operatore di stream

2.6.2 Overloading di `()`

L'operatore `()` è leggermente diverso dagli altri, in quanto è possibile interpretarli in due modi, a seconda che compaiano in un l-value o in un r-value.

In C++ è possibile sovraccaricare gli operatori in modo che si comportino in maniera diversa a seconda della posizione esattamente come accade con gli array.

```
// Operators redefinition
T& operator() (int row, int column) const;
T& operator() (int row, int column);
Matrix<T>* operator* (int k);
```

Figure 2.7: Ridefinizione degli operatori nella classe *Matrix*

Nella nostra applicazione siamo andati a ridefinire appunto l'operatore `()`, come si vede in figura 2.7, in particolare:

- la presenza del simbolo `&` subito dopo il valore ritornato indica che ci sono due definizioni diverse dello stesso operatore.
- il modificatore `const` indica invece che la prima ridefinizione è quella da usare se l'operatore sta a sinistra del simbolo di assegnamento.

Esempi di utilizzo della ridefinizione dell'operatore `()`, li si possono vedere in figura 2.8.

```
(*matrix_a)(0,0) = 8;
cout<<"Get element position (0,0) for MATRIX_A with operator:"<<endl<<(*matrix_a)(0,0)<<endl;
```

Figure 2.8: Esempio di utilizzo della ridefinizione degli operatori

2.7 Templates

Un metodo più potente per il riutilizzo del codice è fornito dai templates. Essi costituiscono uno “schema” di classe che poi verrà realizzato concretamente sostituendo ai tipi parametrici i tipi effettivi dichiarati dagli utilizzatori del template

A differenza del linguaggio Java, ogni differente realizzazione del template costituisce un tipo a sé stante e senza alcuna relazione con gli altri. In Java si può avere una gerarchia di tipi derivati dallo stesso template.

In C++ invece ogni realizzazione di un template viene concretizzata tramite codice indipendente, portando quindi ad un file eseguibile di dimensioni maggiori. Inoltre, per fornire vincoli in modo esplicito come in Java, si deve ricorrere a una “pseudo-ereditarietà”.

Questa flessibilità offerta dai templates, è stata utilizzata nella stesura della classe *Matrix*.

2.8 Standard Template Library

La libreria STL del C++ mette a disposizione una serie di contenitori parametrici, che quindi possono contenere oggetti di tipo arbitrario. Fornisce inoltre iteratori che permettono di visitare tali contenitori e algoritmi per manipolarne gli elementi.

Uno dei contenitori più semplici è *vector*, una classe che permette di gestire collezioni di oggetti ordinati come un array.

I vantaggi del *vector* sul semplice array sono diversi, tra cui la gestione automatica della memoria e la presenza di diversi metodi per le operazioni più comuni quali l'inserimento di un nuovo valore.

La classe *Matrix* utilizza un *vector* per memorizzare gli elementi della matrice ed essendo parametrica di parametro *T*, utilizza un *vector<vector<T>*.

2.8.1 STL - Iterator

Tra le altre funzionalità messe a disposizione da *vector* e dagli altri contenitori in STL ci sono gli iteratori, che costituiscono il modo standard di accedere agli elementi nel contenitore stesso. Gli iteratori si comportano in modo simile a semplici puntatori (ad esempio ammettono l'operatore `++`), anche se in realtà sono più flessibili e possono essere utilizzati anche con le liste.

```

template<class T> Matrix<T>* Matrix<T>::multiply(int k){
    Matrix<T>* result = new Matrix<T>(this);
    typename vector<vector<T>*>::iterator rowsIterator = result->elements->begin();
    while (rowsIterator != result->elements->end()){
        typename vector<T>::iterator columnsIterator = (*rowsIterator)->begin();
        while (columnsIterator != (*rowsIterator)->end()){
            *columnsIterator = *columnsIterator * k;
            columnsIterator++;
        }
        rowsIterator++;
    }
    return result;
}

```

Figure 2.9: STL - Iterator

2.8.2 STL - Algorithm

La libreria STL mette a disposizione anche degli algoritmi generici e riutilizzabili per operare sui contenitori.

Tali algoritmi sono implementati tramite funzioni template che si possono includere nei propri file tramite la direttiva di include.

Un esempio di algoritmo è la funzione *for-each()*, che applica una funzione parametro a tutti gli elementi di un contenitore.

```

template<class T> double Matrix<T>::getMean(){
    AverageCalculator ac;
    typename vector<vector<T>*>::iterator rowsIterator = this->elements->begin();
    while (rowsIterator != this->elements->end()){
        ac = for_each((*rowsIterator)->begin(), (*rowsIterator)->end(), ac);
        rowsIterator++;
    }
    return ac.getMean();
}

template<class T> T Matrix<T>::getMax(){
    list<T>* elements = this->toList();
    T result = *max_element(elements->begin(), elements->end());
    delete elements;
    return result;
}

```

Figure 2.10: STL - Algorithm

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 distributori 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 cliente (oppure tutta la somma di denaro inserita nel caso di prodotto scaduto).
- Mostrare l’elenco dei prodotti disponibili all’interno del distributore (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 struttura definita come un array di Queue (ovvero una matrice), che non fa altro che andare a riprodurre la fisionomia di un distributore reale.

3.2 Gerarchia delle classi

Nella applicazione realizzata sono stati realizzati due gerarchie facendo uso dei *trait*: i *trait* in Scala corrispondono alle interfacce 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).

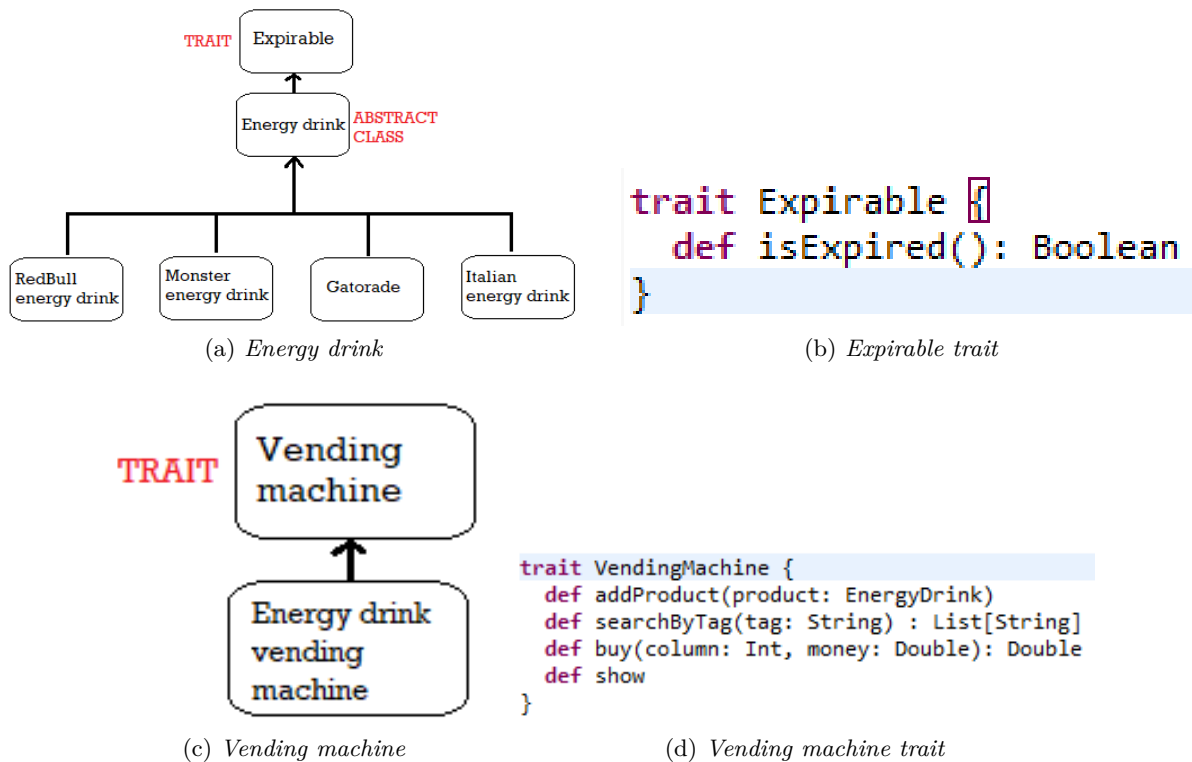


Figure 3.2: Classi e trait

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 elementi 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 infatti 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 assunto da una variabile, tramite un pattern: si tratta in sostanza di una versione leggermente più potente del costrutto *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*, che affiancherà il distributore di bevande energetiche progettato in Scala. In particolare si è definito l'applicazione su una serie di specifiche, quali:

- Il distributore modellato può preparare diversi tipi di bevande (caffè, cappuccino etc), ognuna delle quali è preparata con diverse quantità di ingredienti (acqua, caffè, latte etc) i quali vengono consumati dagli utenti e 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, sempre tenendo 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 4.1, 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 Eventi

La ASM sviluppata modella un sistema event-driven, ovvero un sistema in cui le transizioni da uno stato all'altro sono perlopiù scatenate da input dell'utente, mentre di solito la macchina si trova ferma in uno stato, in attesa di tali eventi. Nel codice della ASM questi eventi sono denominati action; ad ogni stato corrispondono una o più azioni che l'utente può compiere quando la macchina è in quello stato, le quali sono codificate come elementi di un dominio enumerativo, i quali essendo identificativi univoci, non possono essere duplicati, come si vede in figura 4.2.

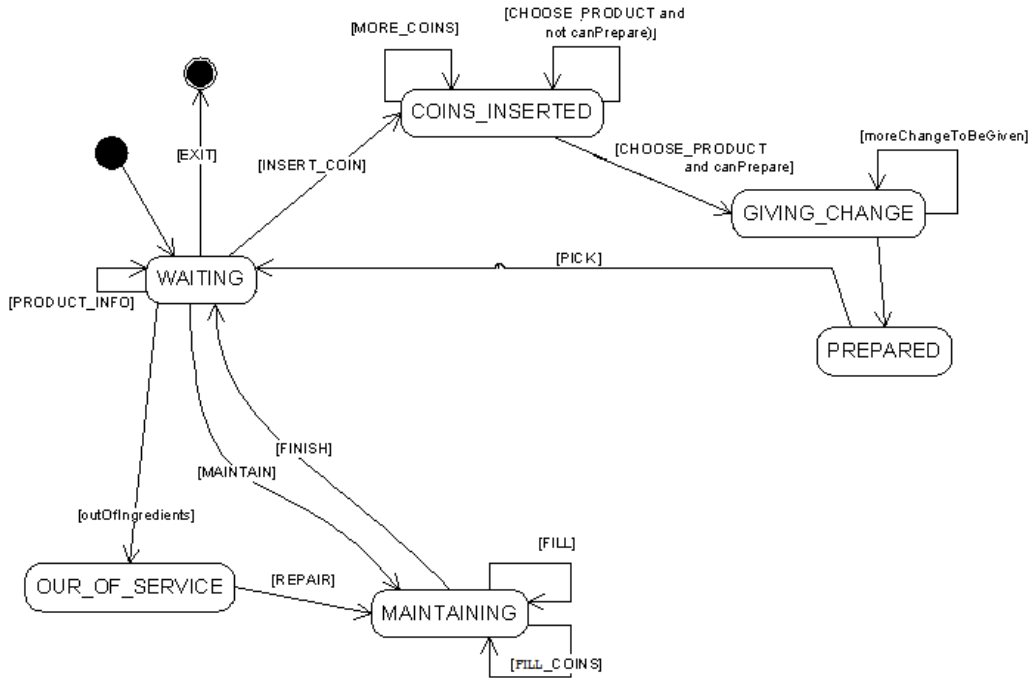


Figure 4.1: Macchina a stati

```
enum domain WaitingAction = {INSERT_COINS | PRODUCT_INFO | MAINTAIN | EXIT}
enum domain CoinsInsAction = {MORE_COINS | CHOOSE_PRODUCT}
enum domain PreparedAction = {PICK}
enum domain OutOfServiceAction = {REPAIR}
enum domain MaintainAction = {FILL | FILL_COINS | FINISH}
```

Figure 4.2: Action che implicano un possibile cambio di stato

La caratteristica event-driven del sistema si è riflessa nella ASM, infatti le regole (*rules*) implementate possono essere suddivise in due categorie:

- regole che attendono il verificarsi di un'azione (qui chiamate event management rules) e poi eseguono le corrette regole di transizione (transition rules)
- regole di transizione che verificano la guardia della transizione e, se verificata, eseguono gli update opportuni

In linea di massima, ad ogni stato corrisponde una *event management rule*, mentre ad ogni arco (transizione) corrisponde una *transition rule*.

4.4 Domini

Sono stati introdotti un dominio enumerativo per gli stati della FSM, uno per ciascun insieme di eventi (ciascun insieme contiene gli eventi validi per uno stato), ovvero quelli relativi alle possibili azioni eseguibili in ogni specifico stato, ed uno relativo alla tipologia di ingredienti utilizzati (figura 4.3)

```
enum domain State = {WAITING | COINS_INSERTED | GIVING_CHANGE | PREPARED | MAINTAINING | OUT_OF_SERVICE}
enum domain WaitingAction = {INSERT_COINS | PRODUCT_INFO | MAINTAIN | EXIT}
enum domain CoinsInsAction = {MORE_COINS | CHOOSE_PRODUCT}
enum domain PreparedAction = {PICK}
enum domain OutOfServiceAction = {REPAIR}
enum domain MaintainAction = {FILL | FILL_COINS | FINISH}
enum domain Ingredient = {COFFEE | MILK | WATER | TEA | CHOCOLATE | PLASTIC_GLASS}
```

Figure 4.3: Domini enumerativi

Sono stati introdotti anche un dominio statico concreto per i tagli di monete riconosciuti dal distributore (rappresentati in centesimi) e un dominio astratto per i prodotti disponibili.

4.5 Controlled - static - monitored functions

4.5.1 Controlled

Le funzioni in figura 4.4, rappresentano funzioni non-monitored così definite:

```
dynamic controlled state           : State
dynamic controlled display        : String
dynamic controlled credit         : Integer
dynamic controlled coinsLeft      : CoinValue -> Integer
dynamic controlled maxSpaceAvailable : CoinValue -> Integer
dynamic controlled quantityLeft   : Ingredient -> Integer
```

Figure 4.4: Funzioni non monitored

Si vede come le prime tre funzioni rappresentino delle funzioni 0-arie, ovvero delle variabili.

Le ultime tre funzioni invece sono n-arie, ovvero mappano dei valori da un dominio ad un codominio: nello specifico associano

- `coinsLeft`: ad ogni `Coins` il numero effettivo di monete presente all'interno del distributore
- `maxSpaceAvailable`: ad ogni `Coins` associa il numero massimo di monete che il distributore può contenere
- `quantityLeft`: ad ogni `Ingredient` un valore `Integer`, che non è altro che la quantità rimasta

I valori assunti dalle funzioni `controlled` rappresentano parte dello stato esteso delle ASM, quindi possono essere utilizzate per mantenere informazioni tra uno stato e l'altro della FSM.

4.5.2 Static

Le funzioni `static` (figura 4.5) sono funzioni la cui interpretazione viene fissata dalla definizione della ASM e non può essere modificata durante l'esecuzione. Possono essere paragonate alle costanti dei linguaggi di programmazione. Per esempio la funzione *price* rappresenta un legame costante tra un

```
static capacity       : Ingredient -> Integer
static quantityNeeded : Prod(Product, Ingredient) -> Integer
static price          : Product -> Integer
```

Figure 4.5: Static functions

prodotto e il suo prezzo.

4.5.3 Monitored

Le funzioni `monitored` rappresentano degli input che l'utente fornisce alla macchina. Il valore di queste funzioni non è persistente, ma viene ad essere specificato dall'utente per ogni stato tramite tastiera (oppure lette anche da file esterno, come fatto per i test automatici tramite file). Tra le funzioni

```
dynamic monitored selectedProduct : Product
dynamic monitored insertedCoin    : CoinValue
dynamic monitored numberOfCoins   : Integer
dynamic monitored filledIngredient : Ingredient
dynamic monitored waitingAction   : WaitingAction
dynamic monitored coinsInsAction  : CoinsInsAction
dynamic monitored preparedAction  : PreparedAction
dynamic monitored outOfServiceAction : OutOfServiceAction
dynamic monitored maintainAction  : MaintainAction
```

Figure 4.6: Monitored functions

`monitored` figurano le funzioni che richiedono all'utente la scelta tra le azioni disponibili. Inoltre ci

sono funzioni con cui l'utente specifica quale moneta, quale prodotto è stato selezionato e, per il manutentore, quale ingrediente è stato rifornito e quante monete ha lasciato nel distributore.

4.6 Inizializzazione

Di default, tutte le funzioni prendono valore undef per quei valori del dominio per cui non sono state esplicitamente definite. Perché la macchina inizi a operare in uno stato diverso, più significativo (anche perché raramente la macchina viene definita in modo da poter gestire valori undef), si deve inizializzare la macchina, ovvero definire uno stato iniziale, come fatto in figura 4.7

```
default init initial_state:
  function state = WAITING
  function display = "Waiting..."
  function coinsLeft ($cv in CoinValue) =
    switch($cv)
      case 5 : 5
      case 10 : 10
      case 20 : 10
      case 50 : 10
      case 100 : 10
      case 200 : 2
    endswitch
  function maxSpaceAvailable ($cv in CoinValue) =
    switch($cv)
      case 5 : 30
      case 10 : 30
      case 20 : 30
      case 50 : 30
      case 100 : 20
      case 200 : 10
    endswitch
  function credit = 0
  function quantityLeft($i in Ingredient) =
    switch($i)
      case PLASTIC_GLASS : 35
      case WATER : 25
      otherwise : 40
    endswitch
```

Figure 4.7: Inizializzazione

In questo caso, lo stato FSM iniziale è quello di attesa, il credito in monete è nullo e la macchina possiede un discreto quantitativo di monete e ingredienti, per poter operare per un certo periodo senza bisogno di manutenzione.

4.7 Event management rules e main rule

Le event management rules si occupano di ricevere le azioni dell'utente e di eseguire (fire) la regola di transizione opportuna. In figura 4.8 c'è un elenco delle regole, la cui struttura è del tutto simile a quella dell'unica regola definita.

La *main rule*, che costituisce l'entry point del programma, controlla per prima cosa che il distributore possa operare, perché non ha finito gli ingredienti (r-selfCheck); questo è possibile grazie al comportamento del blocco sequenziale, che effettua gli update dopo la valutazione di ciascun termine.

Dopo il controllo, invece, vengono valutate in parallelo le regole di gestione degli eventi: per esse non vi è pericolo di *update inconsistenti* perché ciascuna regola ha una guardia che permette di valutare la regola solo quando la macchina si trova nello stato corretto (ogni regola si applica a un diverso stato, quindi sola una alla volta è "attiva").

```

// Deal with events that can happen in the WAITING state
rule r_waitingAction =
  if(state = WAITING) then
    switch(waitingAction)
      case PRODUCT_INFO: r_productInfo[]
      case INSERT_COINS: r_insertCoins[]
      case MAINTAIN:      r_maintain[]
      case EXIT:          skip
    endswitch
  endif

// Deal with events that can happen in the COINS_INSERTED state
rule r_coinsInsAction = ...

// Deal with events that can happen in the PREPARED state
rule r_preparedAction = ...

// Deal with events that can happen in the OUT_OF_SERVICE state
rule r_outOfServiceAction = ...

// Deal with events that can happen in the MAINTAINING state
rule r_maintainAction = ...

```

Figure 4.8: Management rules

4.8 Transition rule

Ad ogni transizione della FSM, anche se rientrando sullo stesso stato, è associata una regola di transizione, che si occupa di valutare la guardia della transizione e, se questa risulta vera, di eseguire gli update opportuni, andando sostanzialmente ad effettuare un update dello stato.

4.9 Simulazione

La macchina è stata simulata con AsmetaS, sia in modalità interattiva che in modalità batch. In modalità interattiva è facile scoprire errori sia di transizione di stato FSM sia di update, perché ad ogni update viene mostrato lo stato completo. Tuttavia una simulazione esaustiva è piuttosto lunga, per cui è difficile trovare errori nelle parti di macchina eseguite più raramente.

La modalità random permette di trovare facilmente violazioni inconsistenti: eseguendo una simulazione random con un numero elevato di transizioni (es. 1000) è più probabile coprire anche situazioni poco frequenti o poco naturali per un utente umano.

Per la simulazione batch è stato creato un file di environment `testCoffee.env` che esegue un tour più o meno completo degli stati e delle transizioni (della FSM).

Per facilitare questo tipo di simulazione è stata introdotta anche l'azione `EXIT` nel dominio `WaitingAction`, che è l'unica azione che produce un update set vuoto; è quindi possibile eseguire la simulazione batch con opzione `-ne`:

```
java -jar AsmetaS.jar -ne -env testCoffee.env CoffeeVendingMachine.asm
```