



Progetti di Informatica III A

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

PIFFARI MICHELE - 1040658

March 23, 2020

Contents

1	I - Cyclone	1
1.1	Introduzione	1
1.2	Descrizione del progetto	1
1.3	Costrutti Cyclone	2
1.3.1	Puntatori @fat e @thin	2
1.3.2	Puntatori @nullable e @notnull	2
1.3.3	Regioni	3
2	II - C++	5
2.1	Descrizione del progetto	5
2.2	Gerarchia delle classi	5
2.3	Multiple inheritance	6
2.4	Diamond inheritance	6
2.5	Costruttori e distruttori	7
2.5.1	Costruttori	7
2.5.2	Distruttori	7
2.5.3	Implementazione nell'applicazione	7
2.6	Overloading particolari	8
2.6.1	Overloading di cout«	8
2.6.2	Overloading di ()	8
2.7	Templates	9
2.8	Standard Template Library	9
2.8.1	STL - Iterator	10
2.8.2	STL - Algorithm	10
3	III - Energy drink vending machine con Scala	11
3.1	Descrizione del progetto	11
3.2	Costrutti utilizzati	12
3.3	Gerarchia delle classi e trait	13
3.4	Filter	14
3.5	Expression oriented programming	15
3.5.1	Match	15
3.5.2	Sealed	15
4	IV - Coffe Machine con ASM	17
4.1	Descrizione del progetto	17
4.2	Macchina a stati	17
4.3	Eventi	18
4.4	Domini	18
4.5	Static and Dynamic functions	19
4.5.1	Static	19
4.5.2	Dynamic	19
4.6	Event management rules e main rule	20

4.7	Inizializzazione	20
4.8	Simulazione	21
5	Correzione prova	23
5.1	EXE 1 - RA	23
5.2	EXE 2 - C	24
5.2.1	Iterativa	24
5.2.2	Ricorsiva senza tail	24
5.2.3	Ricorsiva CON tail	25
5.3	EXE 3 - C++ distruttore	25
5.4	EXE 4 - Opachi	25
5.5	EXE 5 - Visitor	26
5.6	EXE 6 - Scala	26

List of Figures

1.1	@fat pointer visto come una Struct	2
1.2	Gestione pointer nullable	3
2.1	UML class diagram	5
2.2	Esempio di eredità multipla della classe <i>Matrix</i>	6
2.3	Diamond problem	6
2.4	Virtual Inerithance nelle classe <i>Addable</i> e <i>Subtractable</i>	6
2.5	Costruttori classe <i>Matrix</i>	7
2.6	Firma e implementazione del distruttore della classe <i>Matrix</i>	8
2.7	Ridefinizione dell'operatore di stream	8
2.8	Ridefinizione degli operatori nella classe <i>Matrix</i>	9
2.9	Esempio di utilizzo della ridefinizione degli operatori	9
2.10	Utilizzo concreto dei template	9
2.11	STL - Iterator	10
2.12	STL - Algorithm	10
3.1	Prodotti disponibili	12
3.2	Classi e trait	13
3.3	Utilizzo comando <i>filter</i>	14
3.4	Ricerca del tag su ogni possibile prodotto con <i>checkTag</i>	14
3.5	Sealed	15
4.1	Macchina a stati	17
4.2	Action che implicano un possibile cambio di stato, qualora la condizione sia verificata	18
4.3	Domini enumerativi	18
4.4	<i>Signature</i> delle static functions utilizzate	19
4.5	Funzioni dynamic - controlled	19
4.6	Monitored functions	20
4.7	Management rules	20
4.8	Inizializzazione	21
4.9	Test iterativo ASM	21
4.10	Test random ASM	22
5.1	Parziale soluzione esercizio RA	23
5.2	Serie numeri pari Iterativa	24
5.3	Serie numeri pari ricorsiva senza tail [operazioni di debug omesse]	24
5.4	Serie numeri pari ricorsiva senza tail - debug print	24
5.5	Metodo <i>make</i>	25
5.6	Confronto tra metodi calcolaMax (prova sopra e correzione sotto)	26
5.7	Wrap in Scala	26

1

I - Cyclone

1.1 Introduzione

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

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. Si è provato anche il porting semi-automatico, come supporto per la traduzione manuale.

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.

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.

1.2 Descrizione del progetto

L'applicazione scritta in C e successivamente tradotta in Cyclone, consiste in un piccolo programmino in grado di leggere dati da un file, parsarli, fornendo poi una 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, da spedire ai propri cari per gli studenti fuori sede.

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 cartolina 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 di Cyclone 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**)
- **Regioni:** vedi sezione 1.3.3

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

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** (definiti in modo abbreviato con il carattere *f*) effettuano invece tale controllo ogni volta che viene utilizzata l'aritmetica dei puntatori. Essi possono essere pensati come una struttura (struct), 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 visto come una 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 @notnull

@nullable

I puntatori sono di default **@nullable**, ovvero possono assumere valore NULL. Tali puntatori si possono definire anche esplicitamente con **@nullable*. I puntatori fat possono essere solo @nullable: un puntatore @fat@notnull può essere nullo,

Questa tipologia di identificatori per i puntatori sono stati utilizzati nella fase di apertura del file *.txt*: come si vede in figura 1.2, non uso di puntatore *@notnull* per garantire la gestione del caso in cui non si riesca ad aprire il file di testo, tramite un print di errore.

@notnull

I puntatori @notnull 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 sicuramente i più utilizzati, sia perché non introducono l'overhead del controllo di non-nullità (che


```
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

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.

2

II - 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 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.

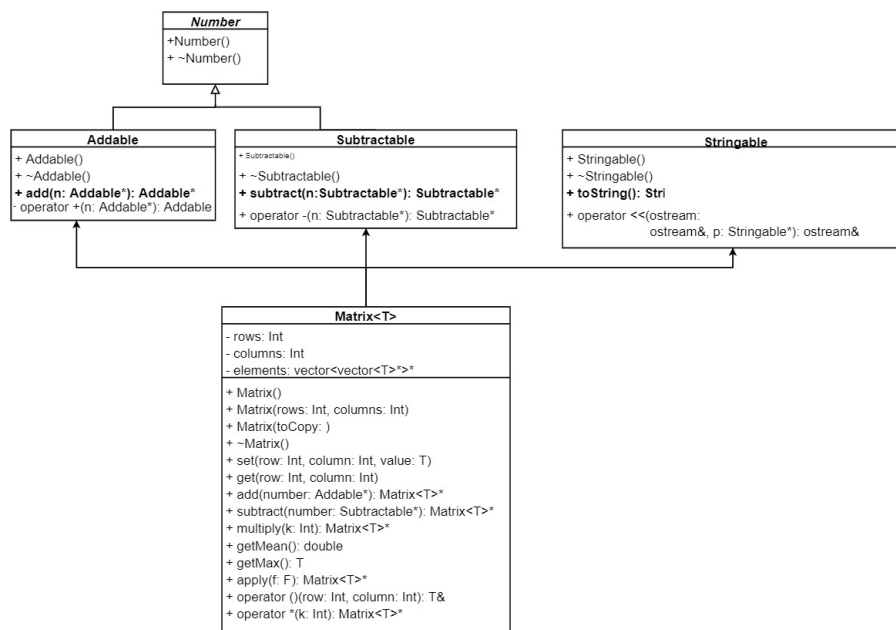


Figure 2.1: UML class diagram

2.3 Multiple inheritance

Nella gerarchia delle classi sopra esposta, 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) i quali permettono di implementare più interfacce ma non di ereditare di più classi.

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 sono pure virtual.

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 quando si attua una doppia ereditarietà da due classi diverse, la quale è una peculiarità del C++) è la possibilità di generare una **gerarchia a diamante**.

Essa 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?

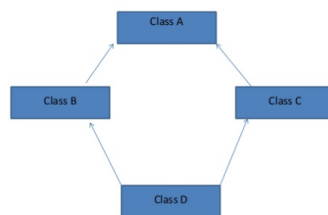


Figure 2.3: Diamond problem

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*

2.5 Costruttori e distruttori

2.5.1 Costruttori

La libertà di poter utilizzare oggetti all'interno delle applicazioni scritte in C++, si incontra però con la necessità di andare a compiere due operazioni preliminari:

- Allocare la memoria dei membri dell'oggetto in esame;
- Inizializzare i membri.

Queste operazioni sono svolte da una particolare funzione membro detta **costruttore**, il cui nome coincide con quello della classe e non presenta alcun valore di ritorno.

In C++ un costruttore che non accetta argomenti, o in cui tutti gli argomenti sono opzionali, è detto **costruttore di default** ed è possibile sovraccaricare questo metodo, fornendo quindi diverse alternative. E' comunque sempre buona norma andare a realizzare l'overloading del costruttore di default, perchè esso ha dei limiti come il fatto che il valore predefinito per il tipo di ogni dato membro può dipendere dal compilatore utilizzato, e quindi non essere deterministico.

2.5.2 Distruttori

Dal lato opposto, il **distruttore** si occupa del compito di rilasciare tutte le risorse associate ad un oggetto, quando finisce la lifetime dell'oggetto stesso: infatti il distruttore viene invocato per tutte le variabili automaticamente ogni volta che viene raggiunta la fine del loro *scope* e tutte le volte che le variabili dinamiche vengono deallocate con la parola chiave *delete*.

In particolare, come per il costruttore, anche il distruttore presenta una firma particolare, ovvero presenta lo stesso nome della classe solamente per il fatto che è preceduto dal simbolo *tilde*; anch'esso, come il costruttore, non presenta alcun valore di ritorno, e in più non presenta parametri (motivo per cui non può essere sovraccaricato).

Inoltre se non viene esplicitamente incluso nella dichiarazione della classe, il distruttore viene generato automaticamente dal compilatore. In questo caso tuttavia, l'unica risorsa rilasciata è la memoria occupata dall'oggetto: se alcuni dei suoi dati membro sono delle reference, o se l'oggetto fa uso di altre risorse (file o altre interfacce di I/O), potrebbe essere necessario svolgere delle operazioni per il loro corretto rilascio.

2.5.3 Implementazione nell'applicazione

Queste caratteristiche sono state sfruttate per quanto riguarda gli oggetti della classe *Matrix*.

Per quanto riguarda la costruzione degli oggetti di tipo *Matrix* sono stati sovraccaricati i costruttori di default, con 2 alternative differenti: come si vede in figura 2.5

```
// Constructors (overloaded)
Matrix(); // Identity matrix 1 x 1
Matrix(int rows, int columns); // Identity matrix rows x columns
Matrix(Matrix<T>* toCopy); // Copy a matrix
```

Figure 2.5: Costruttori classe *Matrix*

In particolare si ha che:

- Il primo costruttore rappresenta il costruttore di default, ed è stato implementato per fornire un'inizializzazione "custom" (e non di default del compilatore) ai membri della classe. Esso va a creare una matrice identità di dimensioni 1x1;
- Il secondo costruttore invece va a inizializzare i membri della classe *Matrix* in maniera tale da ricreare una matrice di dimensioni *rows x columns* (parametri passati in input al costruttore);

- L'ultimo overriding del costruttore di *Matrix* permette di inizializzare un nuovo oggetto *Matrix* tramite un altro oggetto di tipo *Matrix*.

Per quanto riguarda il distruttore (figura 2.6) sono due gli aspetti da porre in evidenza:

- è stato definito *virtual* in maniera tale che venissero chiamati anche i distruttori delle superclassi (come si vede dal log che si ottiene dall'esecuzione del main del progetto);
- esso si occupa, come si vede nella relativa implementazione, di andare a liberare la memoria allocata per la struttura contenente gli elementi della matrice: per fare ciò si è fatto uso di un iteratore sulle righe della matrice, in maniera tale da andare poi a liberare ogni elemento della colonna.

```
virtual ~Matrix(); // Default destructor
template<class T> Matrix<T>::~Matrix(){
    typename vector< vector<T>* >::iterator rowsIterator = this->elements->begin();
    while (rowsIterator != this->elements->end()){
        delete *rowsIterator;
        rowsIterator++;
    }
    delete this->elements;
}
```

Figure 2.6: Firma e implementazione del distruttore della classe *Matrix*

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 2.7, un print personalizzato, per distinguere così il log della libreria da quello di sistema classico.

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

Figure 2.7: Ridefinizione dell'operatore di stream

2.6.2 Overloading di ()

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

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.

Nella nostra applicazione siamo andati a ridefinire appunto l'operatore (), come si vede in figura 2.8, in particolare:

- la presenza del simbolo & subito dopo il valore ritornato indica che ci sono **due definizioni diverse dello stesso operatore**.

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

Figure 2.8: Ridefinizione degli operatori nella classe *Matrix*

- il modificatore **const** indica invece che la **prima ridefinizione è quella da usare se l'operatore sta a sinistra** (*l-value*) del simbolo di assegnamento.

Un esempio di applicazione della ridefinizione dell'operatore (), li si possono vedere in figura 2.9.

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

Figure 2.9: 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.

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 invece avere una gerarchia di tipi derivati dallo stesso template (che non è da confondere con questa casistica: "Dati due tipi concreti A e B per esempio MyClass<A> non ha alcuna relazione con MyClass, a prescindere dal fatto che A e B siano correlate. Il genitore comune di MyClass<A> e MyClass è Object"; ci si riferisce al fatto che si può utilizzare l'invocazione generica anche con i sottotipi, ma tra ogni generic non c'è legame di gerarchia).

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*: nell'applicazione abbiamo implementato matrici di interi (come si vede in figura 2.10), per poter così supportare operazioni numeriche tra di loro.

Nulla vieta ovviamente anche di poter creare matrici di stringhe: questo però richiederebbe un adattamento della libreria (per consentire somma e sottrazioni, qualora definibili, tra stringhe).

```
Matrix<int>* matrix_a = new Matrix<int>(3,3); // Identity matrix 3 x 3
```

Figure 2.10: Utilizzo concreto dei templates

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

- Gestione automatica della memoria;

- Presenza di diversi metodi per le operazioni più comuni quali l'inserimento di nuovi valori.

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: come si vede in figura 2.11, nella nostra applicazione, abbiamo utilizzato **iterator** per andare a deallocare la memoria occupata dai valori che compongono la matrice.

```
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 2.11: STL - Iterator

2.8.2 STL - Algorithm

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

Tali algoritmi sono implementati tramite funzioni template che possono essere inclusi nei propri file tramite la direttiva di *include*.

Esempi di algoritmi utilizzati nel codice sono le funzioni:

- **for-each**, che applica una funzione parametro a tutti gli elementi di un contenitore;
- **max-element**, la quale cerca il massimo valore all'interno di una collezione (nel nostro caso itera da *begin* fino a *end*).

```
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 2.12: STL - Algorithm

3

III - 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, in fase di installazione, per funzionare in una lingua piuttosto che in un'altra: nel codice è gestita solamente la possibilità di introdurre distributori automatici in **lingua italiana** e in **lingua inglese**.
- Ogni distributore è riconosciuto tramite un identificativo univoco (**ID**).
- Questa tipologia di distributori sono in grado di erogare le seguenti bevande energetiche (*energy drink*):
 - **RedBull**;
 - **Monster**;
 - **Gatorade**;
 - **Italian** (particolare energy drink *made in Italy*, che NON viene erogato da distributori in lingua inglese).

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 e gli usi principali di ogni tipologia di energy drink (ovviamente, ogni lattina di RedBull, per esempio, avrà lo stesso insieme di tags).
- **Valore nutrizionale** indicato tramite tre possibili step
 - * **Ipercalorico**
 - * **Normocalorico**
 - * **Ipocalorico**

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).

- **Stampa dell'elenco dei prodotti disponibili** all'interno del distributore, 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 specifica evenienza
- **Aggiunta di lattine** di energy drink all'interno del distributore.

3.2 Costrutti utilizzati

- Traits (sezione 3.3);
- Comando *filter* (sezione 3.4)
- Match e Sealed (sezione 3.5)

3.3 Gerarchia delle classi e trait

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).

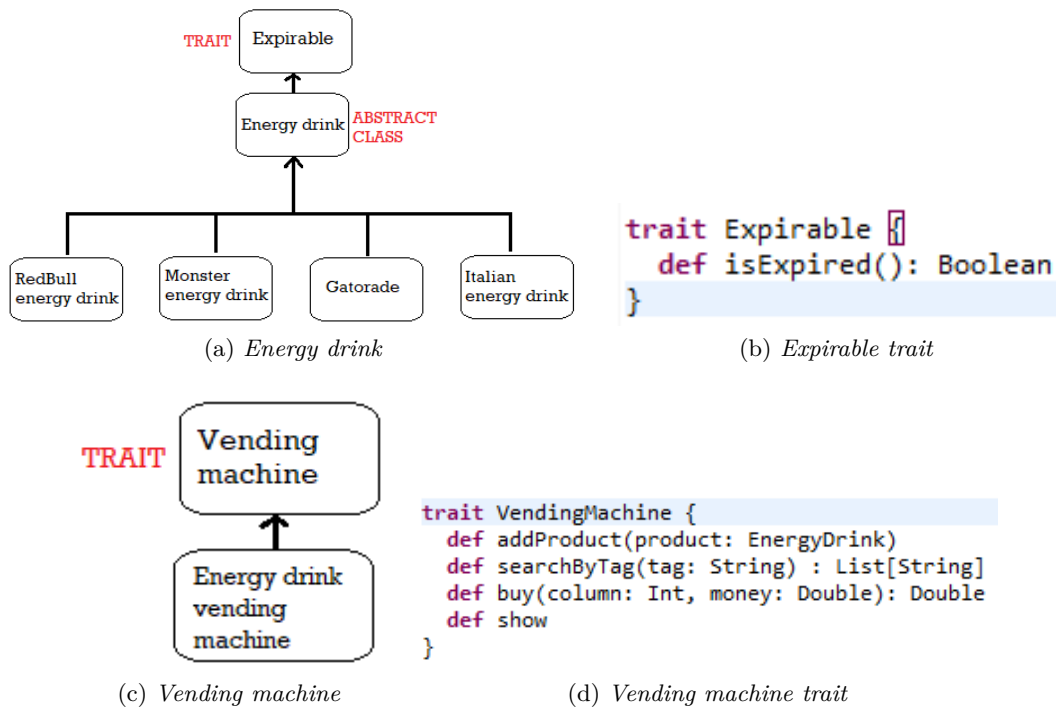


Figure 3.2: Classi e trait

Il primo albero gerarchico rappresenta la struttura che "governa" l'insieme dei possibili energy drink: ogni energy drink astratto eredita da *Expirable* la possibilità di "scadere", che, grazie all'*override* implementa direttamente nell'interfaccia astratta, dato che è uguale per ogni tipologia di energy drink concreta.

Il secondo albero gerarchico invece rappresenta l'appartenenza dell'unica tipologia di distributori trattati in questa applicazione (quelli di energy drink), ad un'interfaccia comune a tutte le possibili tipologie di distributori.

3.4 Filter

Per quanto riguarda la funzionalità di ricerca delle bevande in base ai tag, è stata utile l'istruzione *object oriented filter*: in questo modo l'utente, in base alle diverse necessità (per esempio: studio intenso), può cercare l'energy drink più adatto.

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 crearne una nuova contenente solamente gli elementi che rispettano il criterio di filtraggio definito in maniera custom (questo criterio è passato sotto forma di *closures*, ovvero una funzione definita localmente "al volo").

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*

```
val columns = Vector(RED_BULL, MONSTER, GATORADE, ITALIAN)
var columnWithTagRequested = List[String]()
for(column <- columns) {
  if(!isEmpty(column)) {
    if(checkTag(products(column))()) {
      if(column == 0) {
        columnWithTagRequested = columnWithTagRequested + "Red Bull"
      } else if(column == 1) {
        columnWithTagRequested = columnWithTagRequested + "Monster"
      } else if(column == 2) {
        columnWithTagRequested = columnWithTagRequested + "Gatorade"
      } else {
        columnWithTagRequested = columnWithTagRequested + "Italian"
      }
    }
  }
}
```

Figure 3.4: Ricerca del tag su ogni possibile prodotto con *checkTag*

Come si vede in figura 3.3, in base alla tipologia di lingua con cui il distributore è stato impostato, si va a cercare, **per ogni prodotto disponibile** (figura 3.4), nei rispettivi tag, solamente quelli che contengono la parola chiave cercata dall'utente.

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 (dato che ogni prodotto ha lo stesso set di tags).

Una volta effettuato il filtraggio **ho considerato solamente i prodotti il cui risultato dall'operazione di filter non fosse un vettore vuoto**: questo corrisponde infatti al prendere solamente i prodotti che contengono il tag cercato, notificando poi all'utente la loro tipologia, tramite una stampa a display.

3.5 Expression oriented programming

3.5.1 Match

Il *pattern match* 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.

Nell'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*.

3.5.2 Sealed

Classi e traits possono essere segnati come *sealed*: questo significa che tutti i sotto tipi devono essere dichiarati e implementati all'interno dello stesso file, garantendo che tutti i sottotipi siano conosciuti e noti già in fase di compilazione, prevenendo quindi errori nel nostro codice.

In questo modo, unendo l'utilizzo di classi *sealed* con il *pattern match*, siamo in grado di garantire *type safety*. Infatti questa struttura permette di garantire il fatto che i *match cases* utilizzati saranno tutti esaustivi: questo grazie al fatto il compilatore conosce in anticipo tutte le possibili implementazioni, dato che sono implementate tutte nello stesso file.

```
// Sealed to guarantee that all subclass will be declared in the same file [here]
sealed trait Calorico

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

Figure 3.5: Sealed

IV - Coffe Machine con ASM

4.1 Descrizione del progetto

Per quanto riguarda la parte di *ASM* si è deciso di riprendere un esempio visto in classe (e durante le pause caffè, molto amate da noi studenti), ovvero quello relativo alla *Coffee machine*, che affiancherà il distributore di bevande energetiche progettato in Scala.

In particolare l'applicazione è stata definita basandosi su una serie di specifiche, quali:

- Il distributore modellato può preparare **diverse tipologie 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;
- Il distributore accetta **pagamenti solamente in moneta** tramite l'inserimento di denaro nell'apposita fessura;
- Il distributore è in grado di fornire il resto;
- Quando tutte gli ingredienti sono esauriti, il distributore va fuori servizio, in attesa che gli ingredienti vengano aggiunti dal manutentore (**questa parte non è però stata modellizzata nella presente ASM**).

4.2 Macchina a stati

La ASM è basata su una macchina a stati finiti, mostrata in figura 4.1, che definisce i principali stati e le principali transizioni che si possono verificare durante il funzionamento del distributore.

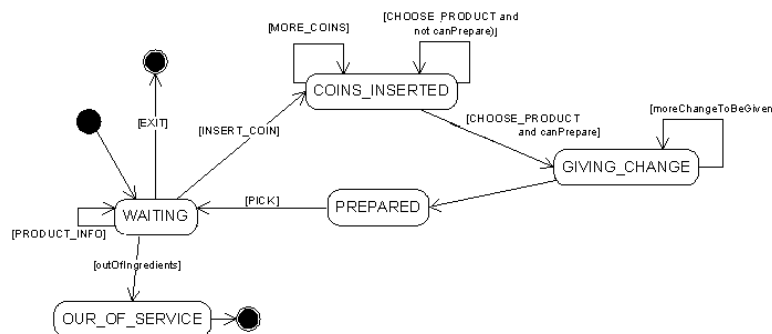


Figure 4.1: Macchina a stati

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 ASMETA questi eventi sono denominati **action**: ad ogni stato corrispondono una o più azioni che l'utente può compiere quando la macchina si trova in quello stato, le quali sono codificate come elementi di un dominio enumerativo (figura 4.2).

```
enum domain WaitingAction = {INSERT_COINS | PRODUCT_INFO | EXIT}
enum domain CoinsInsAction = {MORE_COINS | CHOOSE_PRODUCT}
enum domain PreparedAction = {PICK}
```

Figure 4.2: Action che implicano un possibile cambio di stato, qualora la condizione sia verificata

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**);
- regole di transizione (**transition rules**) che verificano la **condizione** (spesso chiamata anche **guardia**) della transizione e, se verificata, eseguono gli update opportuni per effettuare il passaggio di stato. Esse possono infatti essere viste come istruzioni del tipo

if Condition then Update.

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

4.4 Domini

I domini introdotti nel codice ASM sono:

- **Domini enumerativi**: 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);
- **Dominio statico concreto** per i tagli di monete riconosciuti dal distributore (in centesimi);
- **Dominio astratto** per i prodotti erogabili dal distributore.

```
enum domain State = {WAITING | COINS_INSERTED | GIVING_CHANGE | PREPARED | OUT_OF_SERVICE}
enum domain WaitingAction = {INSERT_COINS | PRODUCT_INFO | EXIT}
enum domain CoinsInsAction = {MORE_COINS | CHOOSE_PRODUCT}
enum domain PreparedAction = {PICK}
enum domain Ingredient = {COFFEE | MILK | WATER | TEA | CHOCOLATE | PLASTIC_GLASS}
```

Figure 4.3: Domini enumerativi

4.5 Static and Dynamic functions

4.5.1 Static

Le funzioni static (figura 4.4) sono funzioni la cui interpretazione viene fissata dalla definizione della ASM e non può essere modificata durante l'esecuzione: questo significa il valore che forniscono come risultato non dipende dagli stati attuali della ASM, e quindi dalla sua dinamica.

Possono essere paragonate alle costanti dei linguaggi di programmazione classici.

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

Figure 4.4: *Signature* delle static functions utilizzate

Per esempio la funzione *price* rappresenta un legame costante tra un prodotto e il suo prezzo, e sarà sempre lo stesso, sia nel caso in cui ci si trova nello stato di "*monete inserite*" che nel caso di stato pari a "*out of service*".

4.5.2 Dynamic

Le funzioni dinamiche, a differenza di quelle statiche, dipendono dallo stato in cui l'ASM si trova, e quindi andranno a fornire una risposta diversa a seconda di quelle che sono le condizioni in cui si trova la macchina a stati. Le funzioni dinamiche sono a loro volta suddivise in 4 sotto-classi: nella applicazione realizzata sono state utilizzate le funzioni di tipo *controlled* e *monitored*.

Controlled

Le funzioni in figura 4.5, rappresentano funzioni dinamiche di tipo **controlled** così definite:

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

Figure 4.5: Funzioni dynamic - controlled

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

Le ultime tre invece sono **funzioni n-arie**, ovvero mappano dei valori (come in una tabella) da un dominio ad un codominio: nell'applicazione abbiamo creato 3 funzioni n-arie presentate di seguito:

- **coinsLeft**: associa ad ogni taglio di moneta il numero attuale di monete presente all'interno del distributore;
- **quantityLeft**: ad ogni tipologia di ingrediente associa 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.

Monitored

Le funzioni monitored rappresentano degli input che l'utente fornisce alla macchina: esse infatti sono funzioni che sono lette ma non sono state aggiornate dalla ASM.

Il valore di queste funzioni non è persistente, ma viene ad essere specificato dall'utente per ogni stato tramite tastiera (oppure letto anche da file esterno, come fatto per i test automatici tramite file).

Tra le funzioni monitored figurano le funzioni che richiedono all'utente la scelta di quale azione deve compiere il distributore tra l'insieme di azioni disponibili.

Inoltre ci sono funzioni con cui l'utente specifica quale moneta, quale prodotto è stato selezionato.

```

dynamic monitored selectedProduct      : Product
dynamic monitored insertedCoin        : CoinValue
dynamic monitored numberOfCoins       : Integer
dynamic monitored waitingAction       : WaitingAction
dynamic monitored coinsInsAction      : CoinsInsAction
dynamic monitored preparedAction      : PreparedAction

```

Figure 4.6: Monitored functions

4.6 Event management rules e main rule

Le event management rules si occupano di ricevere le azioni dell'utente e di eseguire la regola di transizione opportuna.

In figura 4.7 c'è un elenco delle regole, la cui struttura è del tutto simile a quella delle altre regole definite.

```

// 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 = ...

```

Figure 4.7: Management rules

La *main rule*, che costituisce l'entry point del programma, controlla per prima cosa che il distributore possa operare, ovvero che non abbia 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”).

4.7 Inizializzazione

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), deve essere inizializzata, ovvero definirò necessario definire uno stato iniziale, come fatto in figura 4.8

In questo caso:

- lo stato FSM iniziale è quello di attesa (*function state = WAITING*);
- il credito in monete è nullo (*function credit = 0*);
- la macchina possiede un discreto quantitativo di monete e ingredienti (chiamata delle funzioni n-arie dinamiche), per poter operare per un certo periodo senza bisogno di manutenzione.

```

default init initial_state:
  function state = WAITING
  function display = "Waiting..."
  function credit = 0

  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 quantityLeft($i in Ingredient) =
    switch($i)
      case PLASTIC_GLASS : 100
      case WATER : 100
      otherwise : 40
    endswitch

```

Figure 4.8: Inizializzazione

4.8 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. Un esempio di simulazione interattiva è riportata di seguito (figura 4.9): in questo esempio abbiamo provato a prendere un caffè da una distributore che aveva ingredienti sufficienti solamente per un caffè. Come si vede infatti, il distributore è andato, dopo aver erogato il prodotto, in stato *OUT OF SERVICE*, terminando l'applicazione.

```

INITIAL STATE:Product={cappuccino,chocolate,coffee,glassOnly,tea}
Insert a symbol of WaitingAction in [INSERT_COINS, PRODUCT_INFO, EXIT] for waitingAction:
INSERT_COINS
Insert a constant in CoinValue of type Integer for insertedCoin:
20

<UpdateSet - 0>
...
display=Credit: 0.2 €
...
</State 1 (controlled)>

Insert a symbol of CoinsInsAction in [MORE_COINS, CHOOSE_PRODUCT] for coinsInsAction:
MORE_COINS
Insert a constant in CoinValue of type Integer for insertedCoin:
10

<UpdateSet - 1>
...
display=Credit: 0.3 €
...
</State 2 (controlled)>

Insert a symbol of CoinsInsAction in [MORE_COINS, CHOOSE_PRODUCT] for coinsInsAction:
CHOOSE_PRODUCT
Insert a abstract constant in Product for selectedProduct:
coffee

...

Insert a symbol of PreparedAction in [PICK] for preparedAction:
PICK
...
display=Wait the maintainer...
...
state=OUT_OF_SERVICE
run terminated

```

Figure 4.9: Test iterativo ASM

Tuttavia una simulazione esaustiva è piuttosto lunga, per cui è difficile trovare errori nelle parti di macchina eseguite più raramente.

La modalità random permette invece 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.

Nello specifico in figura 4.10 si vede che, nella simulazione random abbiamo 10 stati in cui:

Type	Functions	State 0	State 1	State 2	State 3	State 4	State 5	State 6	State 7	State 8	State 9	State 10
☐ ▲	C state		COINS_INSERTED	GIVING_CHANGE	GIVING_CHANGE	GIVING_CHANGE	PREPARED	WAITING	WAITING	COINS_INSERTED	COINS_INSERTED	COINS_INSERTED
☐ ▲	C credit		100	60	10	0	0	0	0	50	55	105
☐ ▲	C coinsLeft(100)		11	11	11	11	11	11	11	11	11	11
☐ ▲	C display		Credit: 1.0 €	Credit: 1.0 €	Giving change - credit left: 0.1 €	Giving change - credit left: 0.0 €	Prepared.	Waiting...	0.1 €	Credit: 0.5 €	Credit: 0.55 €	Credit: 1.05 €
☐ ▲	M insertedCoin	100	100	100	100	100	100	100	50	5	50	
☐ ▲	M waitingAction	INSERT_COINS	INSERT_COINS	INSERT_COINS	INSERT_COINS	INSERT_COINS	INSERT_COINS	PRODUCT_INFO	INSERT_COINS	INSERT_COINS	INSERT_COINS	
☐ ▲	C quantityLeft(TEA)			39	39	39	39	39	39	39	39	39
☐ ▲	C quantityLeft(MILK)			40	40	40	40	40	40	40	40	40
☐ ▲	C quantityLeft(WATER)			99	99	99	99	99	99	99	99	99
☐ ▲	C quantityLeft(PLASTIC_GLASS)			99	99	99	99	99	99	99	99	99
☐ ▲	C quantityLeft(CHOCOLATE)			40	40	40	40	40	40	40	40	40
☐ ▲	C quantityLeft(COFFEE)			40	40	40	40	40	40	40	40	40
☐ ▲	M coinsAction		CHOOSE_PRODUCT	CHOOSE_PRODUCT	CHOOSE_PRODUCT	CHOOSE_PRODUCT	CHOOSE_PRODUCT	CHOOSE_PRODUCT	CHOOSE_PRODUCT	MORE_COINS	MORE_COINS	
☐ ▲	M selectedProduct		tea	tea	tea	tea	tea	glassOnly	glassOnly	glassOnly	glassOnly	
☐ ▲	C coinsLeft(50)				9	9	9	9	9	10	10	11
☐ ▲	C coinsLeft(10)					9	9	9	9	9	9	9
☐ ▲	M preparedAction						PICK	PICK	PICK	PICK	PICK	
☐ ▲	C coinsLeft(5)									6	6	6

Figure 4.10: Test random ASM

- State0: inserita una moneta da 1€;
- State1: scelto come prodotto il *the*;
- State2: vengono diminuiti gli ingredienti utilizzati (1 TEA e 1 WATER) e viene dato il resto;
- State3: il the costa 0.4€, la macchinetta deve fornire un resto di 0.6€. Inizia erogando una moneta da 50 cent, e diminuisce il quantitativo di monete da 50 cent disponibili;
- State4: restano 10 cent di resto da dare. Fornisce quindi il resto e decrementa le monete da 10 cent disponibili;
- State5: il *the* viene prelevato dall'utente;
- State6: la macchinetta si trova in uno stato di WAITING, e un utente chiede informazioni relativamente al solo bicchiere (GLASS ONLY);
- State7: mentre il display mostra il prezzo del bicchiere (0.1€), l'utente inserisce 50 centesimi;
- State8: l'utente inserisce altri 5 centesimi;
- State9: l'utente inserisce altri 50 centesimi;
- State10: terminazione random run.

Correzione prova

5.1 EXE 1 - RA

- Pulita disposizione dei record di attivazione: disposti lateralmente, per riprendere la crescita verso il basso dello stack;
- Eliminata la colorazione delle variabili che rappresentavano riferimento alla stessa area di memoria (quando si ha passaggio per riferimento, come nel caso della funzione $f(int\ x, int\ \&y)$ in cui il parametro y viene passato per riferimento);

Nella figura 5.1, si vede una visione d'insieme (parziale) della correzione dell'esercizio 1, in cui si è seguita una linea di crescita *verticale*, proprio come la memoria stack: in particolare, la **crescita laterale rappresenta lo sviluppo nel tempo della memoria**, al seguirsi delle varie istruzioni.

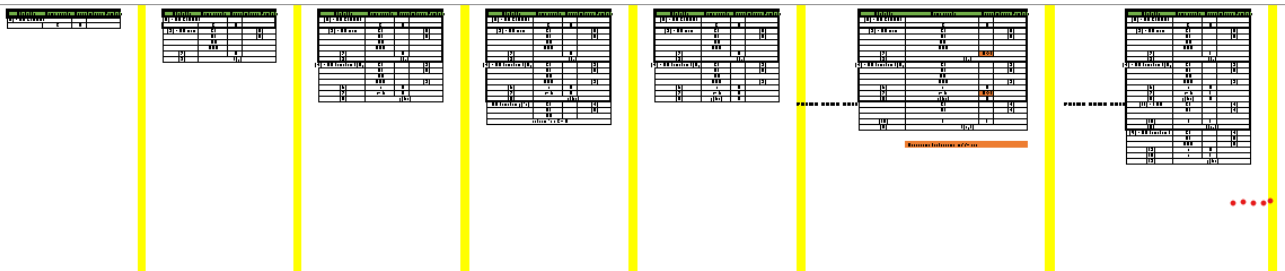


Figure 5.1: Parziale soluzione esercizio RA

5.2 EXE 2 - C

Durante la prova, ho optato per risolvere questo esercizio per ultimo: non è stata una scelta felice, poichè sono arrivato dopo 4 ore di prova a cercare di risolvere un problema all'apparenza complicatissimo ma che, con il senno di poi, si è dimostrato risolvibile (con comunque qualche difficoltà).

5.2.1 Iterativa

Per la parte iterativa, durante la prova, non ci sono stati problemi nella stesura di una soluzione.

Nella figura 5.2, si vede come siamo andati a settare uno spazio di memoria dinamico, con l'istruzione *malloc* di lunghezza sensata: infatti, a tutti i numeri pari da 0 a N ($N/2$) abbiamo aggiunto uno spazio necessario per aggiungere il *numero terminatore* (in questo caso 1).

Il risultato poi è stato popolato con accesso tramite deferenziazione (riga 21).

```

15 int* serieNumeriPari(int N) {
16     int* res = (int*) malloc(sizeof(int) * ((N / 2) + 1));
17     int j = 0;
18
19     for(int i = 1; i <= N; i++) {
20         if(i % 2 == 0) {
21             *(res+j) = i;
22             j++;
23         }
24     }
25
26     *(res+j) = 1;
27     return res;
28 }

```

Figure 5.2: Serie numeri pari Iterativa

5.2.2 Ricorsiva senza tail

Sicuramente questa soluzione richiedeva uno sforzo maggiore.

Riprovando a casa, sono arrivato ad una soluzione, riportata in figura 5.3.

```

40 int* serieNumeriPariRicorsiva_wrapped(int N, int end, int contatoreNumeriPari) {
41     if(N == (end + 1)) {
42         int* i = (int*) calloc(1, sizeof(int));
43         *i = 1;
44         return i;
45     } else {
46         if(N % 2 == 0) {
47             int* res = (int*) malloc((contatoreNumeriPari + 1) * sizeof(int));
48             *res = N;
49             int* temp = serieNumeriPariRicorsiva_wrapped(N + 1, end, contatoreNumeriPari - 1);
50             memcpy(res+1, temp, sizeof(int) * contatoreNumeriPari);
51             free(temp);
52             return res;
53         } else {
54             return serieNumeriPariRicorsiva_wrapped(N + 1, end, contatoreNumeriPari);
55         }
56     }
57 }
58
59 int* serieNumeriPariRicorsiva(int N) {
60     return serieNumeriPariRicorsiva_wrapped(2, N, N/2);
61 }

```

Figure 5.3: Serie numeri pari ricorsiva senza tail [operazioni di debug omesse]

```

Res size 2
N: 6|Memory copying to res [6] value of 1 for a size of 1
Res size 3
N: 4|Memory copying to res [4] value of 6 1 for a size of 2
Res size 4
N: 2|Memory copying to res [2] value of 4 6 1 for a size of 3

```

Figure 5.4: Serie numeri pari ricorsiva senza tail - debug print

Rispetto alla prova, ho apportato questi cambiamenti:

- Ho inserito una funzione wrapper, per nascondere all'utente il fatto che la scansione dei numeri parta da 2 (ho scartato lo 0 e l'1 che sono dispari) arrivando fino a N, così da ritornare un risultato che sia in ordine crescente;
- *contatoreNumeriPari* è utilizzato per poter sfruttare al meglio il comando di memcpy, in maniera tale da copiare/allocare solamente la quantità di memoria necessaria
- Il risultato è salvato in una variabile temporanea per poterne poi fare il free: questo fa sì che i valori numerici vengano analizzati in maniera decrescente (prima annido le chiamate per valori decrescenti, e poi analizzo i risultati in maniera decrescente)
- La condizione d'uscita si ha quando raggiungo il valore successivo al valore passato in input dall'utente (end + 1);
- Essendo che salvo il risultato in una variabile temporanea (riga 49), per poterne poi fare il free, l'analisi dei risultati sarà effettuato in ordine inverso: è per questo motivo che il contatore dei numeri pari parte dal valore completo (N/2) e decresce (e quindi i valori da copiare dovranno essere 1, e via via crescenti);

5.2.3 Ricorsiva CON tail

La versione tail è stata leggermente modificata: anche qui, come nella versione non tail, ho inserito una funzione wrapper, per nascondere all'utente il comportamento a basso livello della funzione.

Nello specifico anche qui faccio partire il conteggio da 2, per farlo concludere quando raggiunge il valore (end + 1).

5.3 EXE 3 - C++ distruttore

Il distruttore è il metodo duale del costruttore: esso serve principalmente ad eliminare gli oggetti della memoria, andando quindi a liberare spazio in memoria.

Il distruttore viene chiamato automaticamente dal compilatore quando la variabile esce dal suo scope. Questa eliminazione automatica però non avviene per i puntatori: quindi se ho un puntatore (allocato tramite la funzione malloc) sarà necessario che sia invocato esplicitamente un comando di free per evitare un memory leakage.

Nel caso di utilizzi di sottoclassi è sempre meglio dichiarare il distruttore virtual così che chiami tutti i distruttori delle superclassi (dalla sottoclasse e poi fino alla superclasse)

5.4 EXE 4 - Opachi

Rispetto alla soluzione proposta in prova, sono andato a snellire il metodo *somma* utilizzando l'init *make* come si vede in figura 5.5.

```

17  coppia_interi_ref somma(coppia_interi_ref c1, coppia_interi_ref c2) {
18
19      // Smart way to make sum
20      return make((c1->A + c2->A) , (c1->B + c2->B));
21
22  /*coppia_interi_ref c = (coppia_interi_ref) malloc(sizeof(struct CoppiaInteri));
23  c->A = c1->A + c2->A;
24  c->B = c1->B + c2->B;
25  return c;*/
26  }

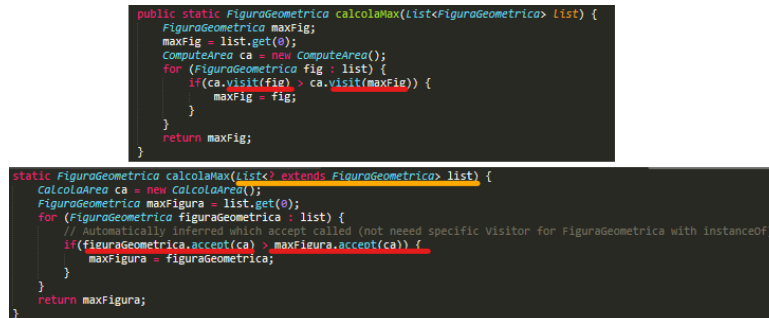
```

Figure 5.5: Metodo *make*

5.5 EXE 5 - Visitor

Durante la prova avevo inserito il Visitor anche per le *FigureGeometriche* astratte: questo perchè nel metodo *calcolaMax* andavo a chiamare la funzione *visit* invece che *accept*, il ch  quindi mi ha portato ad inserire il visitor anche per la classe astratta.

Ho corretto quindi il metodo *calcolaMax* come si vede in figura 5.6.



```

public static FiguraGeometrica calcolaMax(List<FiguraGeometrica> list) {
    FiguraGeometrica maxFig;
    maxFig = list.get(0);
    ComputeArea ca = new ComputeArea();
    for (FiguraGeometrica fig : list) {
        if(ca.visit(fig) > ca.visit(maxFig)) {
            maxFig = fig;
        }
    }
    return maxFig;
}

static FiguraGeometrica calcolaMax(List<FiguraGeometrica> list) {
    ComputeArea ca = new ComputeArea();
    FiguraGeometrica maxFigura = list.get(0);
    for (FiguraGeometrica figuraGeometrica : list) {
        // Automatically inferred which accept called (not need specific Visitor for FiguraGeometrica with instanceof)
        if(figuraGeometrica.accept(ca) > maxFigura.accept(ca)) {
            maxFigura = figuraGeometrica;
        }
    }
    return maxFigura;
}

```

Figure 5.6: Confronto tra metodi *calcolaMax* (prova sopra e correzione sotto)

Ho inoltre corretto anche la segnatura del metodo.

5.6 EXE 6 - Scala

Nella correzione della prova di Scala sono andata ad inserire il wrapper, come si vede in figura 5.7: ho sfruttato la possibilit  che Scala offre di definire funzioni *inline*.

```

def prodPariFino_ricorsivoTail(N: Int): Int = {
    def prodPariFino_ricorsivoTail_wrapped(N: Int, acc: Int): Int = {
        if(N == 1) {
            acc
        } else {
            if(N % 2 == 0) {prodPariFino_ricorsivoTail_wrapped(N - 1, (acc * N))}
            else {
                prodPariFino_ricorsivoTail_wrapped(N - 1, acc)
            }
        }
    }
    prodPariFino_ricorsivoTail_wrapped(N, 1);
}

```

Figure 5.7: Wrap in Scala

Ho inoltre anche aggiunto l'utilizzo delle *High Order Functions*: nella specifico ho inserito una *HOF* per andare a generalizzare il criterio della funzione: nello specifico, con questa *HOF* si pu  andare a settare il valore per cui poi sommeremo solamente i numeri divisibili per tale valore.

Per esempio, nel main di prova, ho settato il criterio pari a 5: in questo modo sommeremo tutti i numeri divisibili per 5.