Relazione dei progetti per l'esame di Informatica III

Fabio Sangregorio

Ingegneria Informatica
Università degli Studi di Bergamo
A.A. 2019-2020

Indice

Elenco delle figure

Elenco dei codici

Capitolo 1

C++

1.1 Descrizione del progetto

Il progetto rappresenta la struttura di un'azienda e in particolare la gestione delle paghe ai vari dipendenti, effettuate sulla base di alcuni parametri. Esso è costituito da quattro tipi di entità legate tra loro:

Stipendiato. possiede un livello di assunzione, il quale determina un moltiplicatore del suo stipendio base.

Salariato. è caratterizzato dal numero di ore lavorate, sulla base delle quali è calcolato il suo salario

Reperibile. riunisce in se sia le caratteristiche di stipendiato che di salariato.

Stagista. ha un rimborso spese fisso

Tutte e quattro estendono (Reperibile in maniera indiretta) la classe astratta Dipendente.

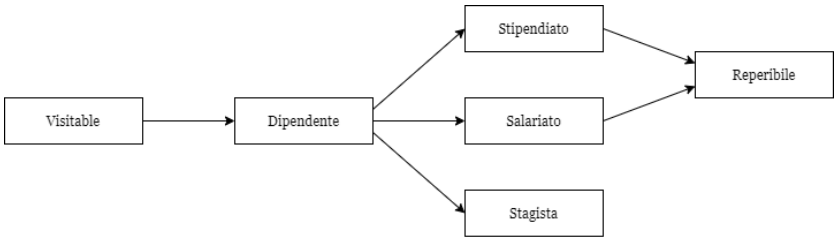

Figura 1.1: C++: rappresentazione delle classi

1.1.1 Integrazione con Cyclone

Il progetto fa uso dell'applicativo Cyclone per la generazione dei file contenenti le paghe dei dipendenti tramite la funzione `system()`, la quale permette di lanciare un comando CLI programmaticamente.

```
1 char *buff = new char[nome.length() + cognome.length() + 100];
2 sprintf(buff, "..\\cyclone\\generapaghe.exe \%s \%s \%f",
  ↪  nome.c_str(),
3                 cognome.c_str(), paga->getValue());
4 cout << buff << endl;
5 system(buff);
```

Listing 1: C++: Integrazione con Cyclone

L'uso corretto del metodo e i parametri dell'invocazione sono spiegati nella sezione **Invocazione** del **Capitolo 2**.

1.2 Classi

Dipendente. Classe astratta che rappresenta un dipendente. Contiene i membri comuni a tutte le sottoclassi.

Stipendiato. Sottoclasse di Dipendente. Rappresenta un classico dipendente a tempo pieno, il quale possiede un livello di assunzione (intero, default: 1) sulla base del quale viene calcolato lo stipendio.

Salariato. Sottoclasse di Dipendente. Rappresenta un dipendente pagato "a cottimo", ossia in base al numero di ore che lavora. E' caratterizzato dal numero di ore lavorate (intero, default: 0)

Reperibile. Sottoclasse sia di Stipendiato che di Salariato. Rappresenta un dipendente assunto a tempo pieno (quindi meritevole di uno stipendio) il quale dà la disponibilità per lavorare un certo numero di ore in reperibilità. Riunisce in se sia le caratteristiche di stipendiato che di salariato. Il suo stipendio è quindi calcolato sulla base del suo livello e delle ore lavorate.

Stagista. Sottoclasse di Dipendente. Rappresenta un dipendente stagista. Ha un rimborso spese fisso, impostabile tramite costante globale.

Visitor. Classe astratta di base per tutti i visitor pattern che si vogliono costruire sulla gerarchia Dipendente. Contiene i metodi virtuali puri `visit`.

Visitable. Classe astratta di base per tutte le classi che vogliono poter essere visitate nel visitor pattern. Contiene il metodo virtuale accept.

Paga. Visitor che visita la gerarchia Dipendente e calcola gli stipendi sulla base dei parametri specificati in precedenza.

1.3 Costrutti C++

1.3.1 Overriding

C++ presenta la caratteristica che, anche se siamo in presenza di ereditarietà e delle condizioni per eseguire l'override di un metodo, esso non viene sempre effettuato, portando a un risultato non sempre polimorfico. In compilazione, infatti, non viene scelta la segnatura ma il metodo da eseguire. Quando si vuole fare override è invece necessario che la classe base dichiari il metodo `virtual`. La `stampaInfo()` nel progetto

è infatti implementato come virtual nella classe base, in modo che nel momento in cui un oggetto di classe figlia esegue questo metodo, esso sceglierà l'oggetto a runtime.

1.3.2 Metodi pure virutal

I metodi `pure virtual` sono utili per poter dichiarare classi astratte in C++. Nel progetto si utilizzano questi metodi nelle classi Visitor, Visitable, e Dipendente.

```
1 virtual void visit (Stipendiato*) = 0;
2 virtual void visit (Salariato*) = 0;
3 virtual void visit (Reperibile*) = 0;
4 virtual void visit (Stagista*) = 0;
```

Listing 2: C++: metodi pure virtual nella classe Visitor

1.3.3 Classi

Tutte le classi fanno uso dei costrutti della programmazione ad oggetti, ossia:

- Costruttori
- Overload dei costruttori
- Distruttori
- Membri e metodi pubblici, protetti e privati

Viene inoltre usato il costrutto `friend` che permette ad una classe di accedere ai membri protetti e privati di un'altra classe.

1.3.4 Ereditarietà multipla

Una caratteristica particolare del C++ è la possibilità per una classe di poter ereditare contemporaneamente da più classi. Essa però

può portare a problemi nel caso le due classi da cui si eredita abbiano una stessa classe base: in questo caso ci si troverebbe con due istanze della stessa classe base. Questo in letteratura è conosciuto come *Diamond problem*. Nel progetto si ha questa struttura con la gerarchia di Dipendente, Salariato, Stipendiato e Reperibile.

```
1 class Dipendente: public Visitable { }
2
3 class Stipendiato: public virtual Dipendente { }
4
5 class Salariato: public virtual Dipendente { }
6
7 class Reperibile: public virtual Salariato, public virtual
  ↪  Stipendiato { }
```

Listing 3: C++: ereditarietà multipla

Il problema si risolve con l'ereditarietà virtuale, implementata nel progetto. Infatti è sufficiente che Stipendiato e Salariato ereditino virtualmente da Dipendente perchè si garantisca che ci sia al più un'istanza di Dipendente attiva.

1.3.5 STL

Si sono voluti usare alcuni costrutti della STL nel software per ottenere funzionalità avanzate senza doverle riscrivere da zero.

unordered_map

Usato da Azienda per mantenere una mappa delle persone. La chiave della mappa è l'id del dipendente, mentre il valore è il puntatore al Dipendente.

Iteratori

Sono stati usati gli iteratori per ciclare sulla mappa dei dipendenti di Azienda.

```
1  void Azienda::generaPaghe() {
2          Paga *paga = new Paga();
3
4          cout << endl << "Generazione paghe:" << endl;
5          for (unordered_map<long, Dipendente*>::const_iterator i =
6                          dipendenti.begin(); i != dipendenti.end();
                            ↪  ++i) {
7                  Dipendente *d = i->second;
8                  d->accept(paga);
9          ...
10         }
11
12         cout << "Paghe generate." << endl;
13 }
```

Listing 4: C++: STL iterators

transform

E' stato usato l'algoritmo `transform` di STL per rendere tutti i caratteri minuscoli di una strtinga prima di effettuarne la ricerca, in modo da rendere la ricerca case insensitive.

```
for (unordered_map<long, Dipendente*>::const_iterator i =
                dipendenti.begin(); i != dipendenti.end(); ++i) {
        string cognome_l = i->second->getCognome();

        std::transform(cognome_l.begin(), cognome_l.end(),
        ↪  cognome_l.begin(),
                        ::tolower);

        if (cognome_l.find(str_l) != std::string::npos) {
                cout << i->first << "\t" << i->second->getNome() <<
                ↪  " "
                                << i->second->getCognome() << endl;
                found = true;
        }
}
```

Listing 5: C++: STL transform

1.4 Pattern

Nel progetto sono stati utilizzati una varietà di design pattern tipici della programmazione ad oggetti.

1.4.1 Singleton

Siccome il software rappresenta la gestione del personale di un'unica azienda, la classe Azienda presenta il pattern Singleton: ha un costruttore privato, un membro statico privato rappresentate l'istanza della classe, e un metodo pubblico `getInstance()` per il suo accesso. In questo modo non è possibile la creazione di più istanze di Azienda. Nel caso si provasse a creare una nuova Azienda, il pattern farà in modo di restituire l'istanza già creata.

```
1  // azienda.h
2  class Azienda {
3  public:
4          static Azienda *getInstance();
5
6  private:
7          static Azienda *instance;
8          Azienda();
9          Azienda(Azienda const& other);
10         void operator= (Azienda const& other);
11 };
12
13 //azienda.cpp
14 Azienda *Azienda::instance = NULL;
15
16 Azienda* Azienda::getInstance() {
17         return instance ? instance : (instance = new Azienda());
18 }
```

Listing 6: C++: Pattern Singleton

Inoltre, anche il costruttore di copia e l'operatore = sono stati ridefiniti ma non implementati, in modo da non permettere la copia dell'istanza singleton.

1.4.2 Getter/Setter

Tutte le classi presentano il pattern getter/setter per la definizione dei membri della classe, in modo da poter definire azioni specifiche all'assegnamento e all'accesso dei campi. Si può veedere un esempio di questo pattern nella classe Dipendente:

```
1 string const& Dipendente::getNome() const {
2         return nome;
3 }
4
5 void Dipendente::setNome(string const& nome) {
6         this->nome = nome;
7 }
```

Listing 7: C++: Pattern Getter/Setter

1.4.3 Visitor

Il Visitor pattern permette di effettuare il double dispatching (e quindi di invocare metodi diversi non solo in base al tipo effettivo del chiamante ma anche a quello del chiamato), e permettono di creare funzionalità sulle classi senza modificarne il codice interno. Nello specifico, nel progetto viene usato questo pattern per calcolare lo stipendio dei vari dipendenti in base al tipo di dipendente. Il cambio di meccanismo sulla quale vengono calcolate le paghe comporterebbe la riscrittura del codice in ogni classe, mentre con l'utilizzo dei Visitor possiamo avere tutta la specifica nella stessa classe.

Anche la classe Visitor è dichiarata `friend` delle classi nella gerarchia di Dipendente, in modo da poter trarre vantaggio della potenza dei Visitor senza dover rinunciare al principio dell'incapsulamento (ossia senza dover settare tutti i metodi a `public`).

Per ottenere questo funzionamento sono state create due classi che funzionano da interfaccia:

```
1  //visitor.h
2  class Visitor {
3  public:
4          virtual ~Visitor() {}
5          virtual void visit (Stipendiato*) = 0;
6          virtual void visit (Salariato*) = 0;
7          virtual void visit (Reperibile*) = 0;
8          virtual void visit (Stagista*) = 0;
9  };
10
11 //visitable.h
12 class Visitable {
13 public:
14         virtual ~Visitable() {}
15         virtual void accept (Visitor* visitor) = 0;
16 };
```

Listing 8: C++: Pattern Singleton

La classe che lo estende deve implementare il metodo sempre allo stesso modo:

```
virtual void accept(Visitor* visitor) { visitor->visit(this); }
```

I visitor dell'applicativo calcolano la paga dei vari tipi di dipendenti nel seguente modo:

```
1  double const PAGA_BASE = 1500;
2  double const PAGA_ORARIA = 10;
3  double const RIMBORSO_SPESE = 400;
4
5  void Paga::visit(Stipendiato* s) {
6          setValue(PAGA_BASE * s->getLivello());
7  }
8
9  void Paga::visit(Salariato* s) {
10         setValue(PAGA_ORARIA * s->getOreLavorate());
11 }
12
13 void Paga::visit(Reperibile* s) {
14         setValue(PAGA_BASE * s->getLivello() + PAGA_ORARIA *
           ↪  s->getOreLavorate());
15 }
16
17 void Paga::visit(Stagista* s) {
18         setValue(RIMBORSO_SPESE);
19 }
```

Listing 9: C++: Calcolo della paga

1.4.4 Facade

Azienda è un Facade per le classi Stipendiato, Salariato, Reperibile e Stagista, in modo da garantire l'accesso a metodi complessi che fanno riferimento a queste classi in modo semplice e nascondendo la complessità delle funzionalità. Il pattern è implementato grazie al qualificatore `friend` di Azienda per le classi citate, in modo da permetterne il controllo della creazione. Siccome i costruttori delle classi sono protected, l'unico modo per crearne delle istanze è passando dalla classe Azienda, la quale assegna un id univoco e ne aggiunge l'istanza alla sua lista di dipendenti.

1.4.5 Template

Per specificare il tipo di ritorno del visitor è stato usato il Template Pattern. Si crea un wrapper alla classe Visitor che contiene anche un campo value, al cui interno è presente il valore di ritorno del visitor. Esso può essere acceduto dall'oggetto che ha invocato il visitor per ottenerne il valore:

```
1  template <typename ReturnType>
2  class ReturnVisitor: public Visitor {
3  public:
4          ReturnType const& getValue() { return value; }
5
6  protected:
7          void setValue(ReturnType const& value) { this->value =
           ↪  value; }
8
9  private:
10         ReturnType value;
11 };
```

Listing 10: C++: Pattern Template

Capitolo 2

Cyclone

2.1 Descrizione del progetto

Il progetto sviluppato riprende il contesto del software scritto in C++, ossia l'ambiente di gestione delle paghe di un'azienda. E' pensato come strumento di supporto al software precedente, in quanto è sostanzialmente un tool che vuole simulare la generazione automatica di un file che rappresenta una busta paga lavorativa.

2.2 Funzionalità

Vengono di seguito mostrate le funzionalità dell'applicativo realizzato.

2.2.1 Funzione principale

genera_paga
Descrizione: genera un file di paga nella cartella corrente, contenente le informazioni della persona associata alla paga e il suo importo.
Parametri:

- *const char * @notnull @fat nome*: nome della persona di cui generare la paga

- *const char * @notnull @fat cognome*: cognome della persona di cui generare la paga
- *double importo*: importo della paga

Ritorno: *int* 0 se l'operazione e' andata a buon fine, 1 altrimenti

Il file generato ha come nome `paga_nome_cognome_data.txt`, con `data` la data in cui è stato generato il file. Il filename è inoltre in maiuscolo.

Di seguito un esempio di output del software:

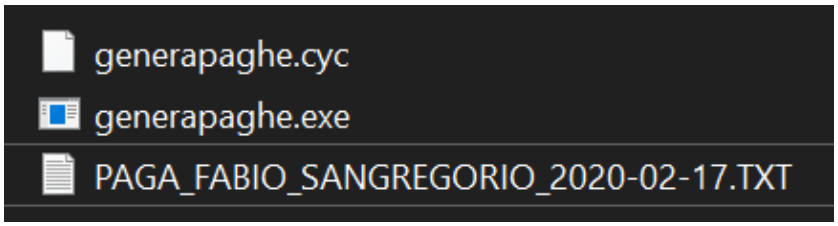

Figura 2.1: Cyclone: esempio di filename

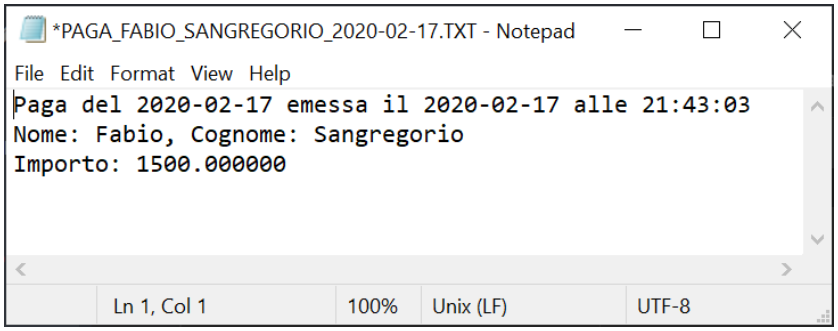

Figura 2.2: Cyclone: esempio di paga

2.2.2 Funzioni di utility

- ***today_to_str***

  Descrizione: copia nella stringa passata come parametro la data e ora attuale

  Parametri:

  - *char * @notnull @fat str*: la stringa in cui copiare il risultato

  Ritorno: *void*

- ***upcase***
  Descrizione: ritorna una nuova stringa con il contenuto della stringa passata come parametro in maiuscolo
  Parametri:

  - *const char * @fat str*: la stringa il cui contenuto rendere maiuscolo

  Ritorno: *char * @zeroterm @notnull @fat* una nuova stringa con il contenuto di str in maiuscolo

2.3 Costrutti Cyclone

Vengono di seguito mostrati i costrutti Cyclone mostrati nel progetto.

2.3.1 Puntatori

Nel software vengono usati i classici puntatori *, con le seguenti differenze rispetto al C:

- Ad ogni dereferenzazione del puntatore viene controllato da Cyclone se il puntatore non è nullo, prevenendo `Segmentation Fault`
- Viene vietato il cast da intero a puntatore, prevenendo errori `Out Of Bound`
- E' vietata l'aritmetica ei puntatori, in modo da prevenire `Buffer Overflow`

2.3.2 Qualificatore @notnull

Il qualificatore `@notnull` indica a Cyclone di controllare all'assegnamento (invece che ad ogni dereferenzazione) che il puntatore non sia nullo, e in caso vietarne l'utilizzo.
Può essere espresso sia con `* @notnull` che semplicemente con `@`.

2.3.3 Qualificatore @zeroterm

Il qualificatore `@zeroterm` è usato per indicare che l'array a cui punta è terminato dal carattere `\0`. E' quindi utile per la gestione delle stringhe.

2.3.4 Qualificatore @fat

Il qualificatore `@fat` permette al puntatore su cui esso viene applicato di mantenere informazioni sul numero degli elementi dell'array, accessibile utilizzando la funzione `numelts(ptr)`. Questo permette l'aritmetica dei puntatori con controllo che non si vada oltre la dimensione dell'array.

Nell'esempio seguente viene utilizzato `numelts` per eseguire una calloc della dimensione adatta e per poi ciclare sui singoli caratteri della stringa, in modo da renderli maiuscoli.

```
1 char * @zeroterm @notnull @fat upcase(const char * @fat str){
2   char * @fat @zeroterm upcased = calloc(numelts(str),
    ↪  sizeof(char));
3   for(int i = 0; i < numelts(str); i++){
4     // trasformazione dei caratteri in maiuscolo
5   }
6   return upcased;
7 }
```

Listing 11: Cyclone: utilizzo del qualificatore @fat

Può essere espresso sia con `* @fat` che semplicemente con `?`.

2.4 Invocazione

Il software può essere invocato tramite Command-line Interface tramite il seguente comando:

```
./generapaghe.exe param1 param2 param3
```

con i seguenti parametri:

- **param1** (stringa): nome dell'utente di cui generare la paga
- **param2** (stringa): cognome dell'utente di cui generare la paga
- **param3** (float): importo della paga

Nel caso manchi uno dei precedenti argomenti verrà mostrato a schermo un messaggio mostrante l'uso corretto del comando. Il software ritorna 0 se la generazione della paga è andata a buon fine, 1 altrimenti. Questo software viene invocato dal progetto C++ tramite una chiamata `system()` in modo da generare i file di paga programmaticamente.

Capitolo 3

Scala

3.1 Descrizione del progetto

Il progetto mostra la soluzione portata al round di qualificazione online del Google Hash Code 2020, realizzata in collaborazione con altri tre compagni di corso: Conti Lorenzo, Ferri Samuele e Gotti Steven.

Il problema consiste nella schedulazione della scannerizzazione e invio di una serie di libri da parte di librerie in modo da massimizzare il punteggio ottenuto da tutti i libri, tenendo in considerazione che ogni libreria deve registrarsi prima di poter inviare i libri.

3.1.1 Libri

Esistono **B** libri diversi con id da 0 a **B**-1. Più librerie possono possedere lo stesso libro, ma esso deve essere scannerizzato solo una volta. Ogni libro è descritto da un parametro, ossia il punteggio assegnato quando esso viene scannerizzato.

3.1.2 Librerie

Esistono **L** diverse librerie con id da 0 a **L**-1. Ogni libreria è descritta dai seguenti parametri:

- il set di libri nella libreria
- il tempo in giorni di registrazione per l'invio
- il numero di libri che possono essere scannerizzati e inviati al giorno dalla libreria, una volta che essa è registrata

3.1.3 Tempo

Ci sono **D** giorni dal giorno 0 a **D**-1. La prima registrazione può iniziare al giorno 0 e **D**-1 è l'ultimo giorno in cui i libri possono essere inviati.

3.1.4 Registrazione della libreria

Solo una libreria alla volta può effettuare il processo di registrazione, quindi una libreria può registrarsi solo se nessun altro processo di registrazione è in esecuzione.

I libri in una libreria possono essere scannerizzati appena la registrazione è stata effettuata, ossia il giorno immediatamente successivo alla registrazione. Essi possono essere scannerizzati in parallelo da più librerie.

Il processo è riassunto nella figura seguente:

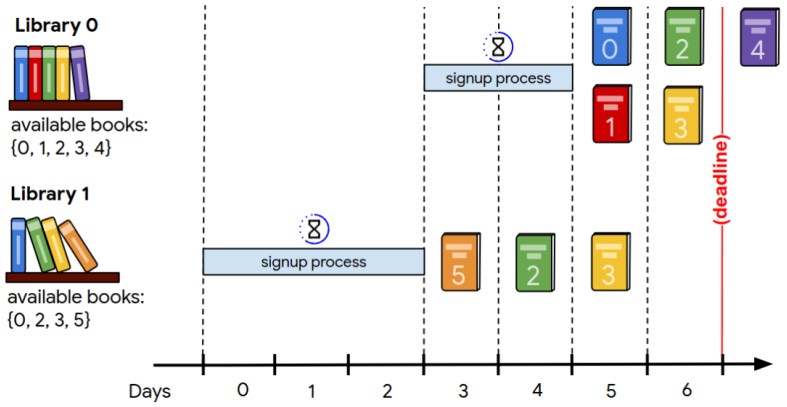

Figura 3.1: Scala: problema

3.2 Costrutti Object Oriented

Nel progetto sono stati usati i seguenti costrutti Object Oriented.

3.2.1 Classi

Library

Rappresenta un oggetto Libreria. La classe possiede i seguenti attributi:

- **id:** l'id della libreria
- **books:** una lista di oggetti di tipo **Book** rappresentante il set di libri nella libreria
- **signupDays:** il tempo di registrazione della liberia necessario all'invio
- **booksPerDay:** numero di libri che possono essere inviati al giorno
- **booksToSend:** lista dei libri che effettivamente verranno inviati dalla libreria. Inizialmente è una lista vuota e verrà riempita man mano che si decide quali libri inviare
- **numBooks:** rappresenta il numero totale di libri nella libreria. E' inizializzato durante l'istanziazione dell'oggetto tramite il corpo del costruttore

Essa possiede anche due metodi, strettamente collegati, usati nell'ordinamento delle librerie.

```scala
def libraryScore(): Int = {
    val thisBooksScore = this.books.foldLeft(0)(_ + _.score)
    this.booksPerDay * thisBooksScore / this.signupDays
}

def compare(that: Library): Int = this.libraryScore -
    that.libraryScore
```

Listing 12: Scala: Metodi di ordinamento delle librerie

L'ordinamento è basato su un rapporto degli attributi della libreria, non applicabili ugualmente su tutti i dataset. In generale, se una libreria è in grado di inviare più libri al giorno, o se i libri stessi hanno un punteggio alto, essa dovrà essere schedulata per prima. Un altro fattore sono i giorni di registrazione: se infatti una libreria ha un tempo di signup basso, è meglio che venga schedulata prima in modo che possa iniziare subito ad inviare i libri.

Book

Rappresenta un singolo libro. Possiede solo due attributi:

- **Id:** l'id del libro
- **score:** il punteggio di tale libro

Anch'esso implementa il metodo `compare` per l'ordinamento dei libri secondo il loro punteggio.

3.2.2 Traits

I traits sono usati per condividere interfacce e campi tra classi. Sono simili alle intefacce di Java 8. Classi e oggetti possono estendere traits, ma i essi non possono essere istanziati e quindi non hanno parametri.

E' stato utilizzato il trait *Ordered* per l'ordinamento di librerie e libri, come mostrato di seguito:

```
class Library(var id : Int, var books : List[Book], var signupDays :
↪  Int, var booksPerDay : Int) extends Ordered[Library] {
    ...
}

class Book(var id: Int, var score: Int) extends Ordered[Book] {
    def compare(that: Book): Int = this.score - that.score
}
```

Listing 13: Scala: Implementazione del trait Ordered

3.2.3 Overriding

E' stato inoltre fatto uso dell'override in modo da definire il metodo *toString()* sulle librerie, come segue:

```
override def toString = id.toString + " :" + "NumBooks: " + numBooks
↪  + ", SignUp: " + signupDays + ", BxD: " + booksPerDay
```

Listing 14: Scala: Uso di override per il metodo toString

3.3 Costrutti funzionali

Nel codice sono stati utilizzati vari costrutti funzionali visti a lezione, di cui vengono riportati degli esempi.

```scala
// Uso di map
val library_books = libraries_info(i+1).split("\\s").map(x => new
 ↪  Book(x.toInt, scores(x.toInt)))

// Uso di foldLeft
def libraryScore(): Int = {
    val thisBooksScore = this.books.foldLeft(0)(_ + _.score)
    this.booksPerDay * thisBooksScore / this.signupDays
}

// Uso di for e foreach
for(library <- res_libraries) {
    writer.append(library.id + " " + library.booksToSend.size +"\n")
    library.booksToSend.foreach(book => writer.append(book.id + "
     ↪  "))
    writer.append("\n")
}

// Uso di filter
res_libraries = res_libraries.filter(_.booksToSend.nonEmpty)
```

Listing 15: Scala: Uso di costrutti funzionali

Nel primo esempio viene mostrato l'uso del costrutto `map` nella manipolazione dell'input per la creazione di oggetti Libro a partire da una lista di righe di testo contenenti le loro informazioni, in modo da costruire la rappresentazione oggettuale dell'informazione direttamente durante il parsing del documento.

Il secondo esempio mostra l'utilizzo di `foldLeft` per sommare i punteggi dei libri contenuti nella libreriain modo da calcolare lo score della stessa. Ciò è usato nel metodo `compare` in modo da ordinare le librerie secondo il loro punteggio.

Il terzo esempio contiene un doppio uso dei cicli, uno con *for comprehension* (`for(library <- res_libraries)`) e uno funzionale (`foreach`).

L'ultimo esempio mostra l'utilizzodi filter, nel contesto di rimuovere

dalla lista dei risultati le librerie che non hanno libri da inviare, prima di scriverle nel file di output.

Capitolo 4

Abstract State Machines

4.1 Descrizione del progetto

La Abstract State Machine realizzata vuole modellizzare un casello autostradale, rappresentandolo attraverso tre stati.

La macchina deve rispettare le seguenti specifiche:

- Si immagina di classificare i biglietti autostradali in 4 gruppi, ciascuno con un valore monetario diverso.
- La sbarra del casello deve essere alzata solo quando è stata inserita una quantità di denaro sufficiente all'acquisto di un biglietto.
- La sbarra deve chiudersi dopo che l'automobile è passata oltre il casello.

4.2 Macchina a stati

In base alle specifiche descritte in precedenza, si è dapprima descritta la macchina a stati del progetto usando il formato standard delle macchine

a stati, per mezzo di UML. Essa è rappresentata schematicamente di seguito:

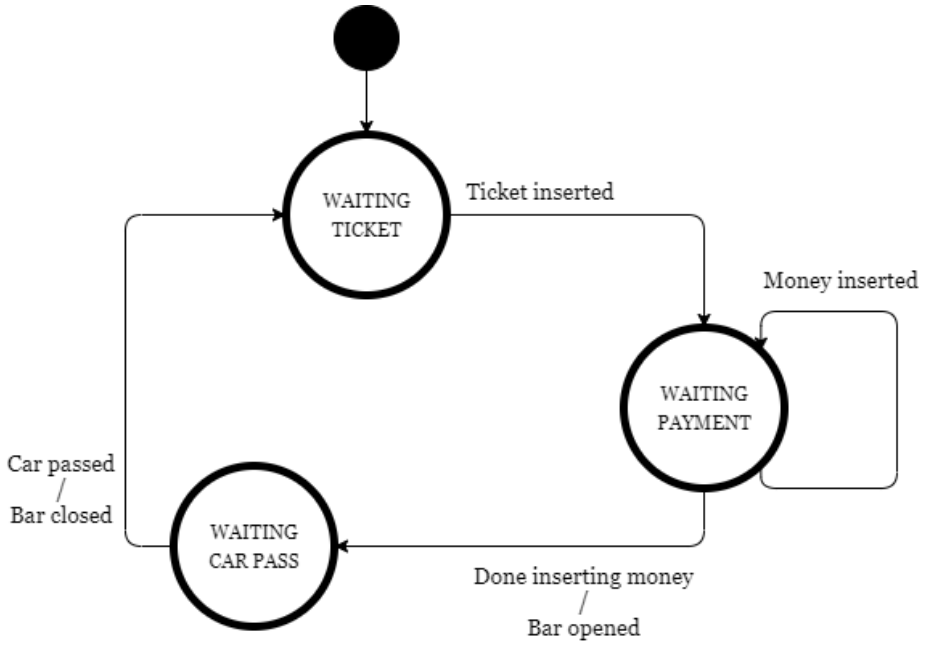

Figura 4.1: ASM: macchina a stati

4.2.1 Stati

Lo stato iniziale della macchina è *WAITING TICKET*. Nel momento in cui un autista giunge al casello e inserisce il ticket, la ASM passa allo stato di *WAITING PAYMENT*, stato in cui resta finche non è stato pagato l'intero importo del biglietto. Al termine del pagamento, l'utente preme il tasto di completamento e, previa verifica che l'importo versato sia una quantità sufficiente al pagamento del biglietto, si passa allo stato di *WAITING CAR PASS* e la sbarra viene aperta. Quando la fotocellula rileva il passaggio dell'automobile, la sbarra viene riabbassata e si torna allo stato iniziale di *WAITING TICKET*.

4.3 Domini

I domini utilizzati per questo progetto sono mostrati di seguito:

```
1 domain Tickets subsetof Natural
2 enum domain Status = { WAITING_TICKET | WAITING_PAYMENT |
  ↪ WAITING_CAR_PASS }
3 enum domain Bar = { OPENED | CLOSED }
```

Listing 16: ASM: Domini

Tickets. è un subset di Natural. E' utilizzato per il pagamento (si assume che i pedaggi abbiano solo importi interi per semplicità).

Status. è un enum. Rappresenta tutti gli stati possibili della macchina.

Bar. è un enum. Rappresenta la posizione della sbarra (aperta o chiusa)

4.4 Funzioni dinamiche

La keyword `dynamic` indica che il valore della funzione può cambiare con il tempo, rappresentando quindi in pratica una variabile.

4.4.1 Funzioni controllate

Le funzioni controlla te sono funzioni che posson oessere lette e scritte solo dalla macchina a stati. Sono indicate dalla keyword `controlled`.

Nel progetto sono state utilizzate le funzioni controllate 0-arie *toPay*, che indica quanto è l'importo da pagare, *balance* che indica quanto è stato versato, e *gateStatus* che indica in quale stato si trova attualmente la macchina.

```
1 dynamic controlled toPay : Natural
2 dynamic controlled balance : Natural
3 dynamic controlled gateStatus : Status
```

Listing 17: ASM: Funzioni controllate

4.4.2 Funzioni monitorate

Le funzioni monitorate possono essere scritte solo dall'environment, ma possono essere anche lette dalla macchina a stati. Si utilizza questo tipo di variabili per inserire comandi e valori nella macchina a stati dall'esterno. Sono contrassegnate dalla keyword `monitored`.

Nel progetto sono state utilizzate le funzioni monitorate 0-arie *ticketInserted* e *moneyInserted*, le quali rappresentano rispettivamente il biglietto inserito e i contanti inseriti.

```
1 dynamic monitored ticketInserted : Tickets
2 dynamic monitored moneyInserted : Natural
```

Listing 18: ASM: Funzioni monitorate

4.4.3 Funzioni output

Le funzioni 0-arie di output utilizzate nella macchina sono *bar*, la quale rappresenta lo stato della sbarra, e *message* che contiene il messaggio visualizzato sul display.

4.5 Funzioni statiche

4.5.1 Funzioni derivate

Le funzioni derivate sono costanti che derivano da funzioni le quali assegnano staticamente i loro valori, ma possono variare per via degli

argomenti. La funzione *ticketPrice* è una funzione unaria che dato il numero di ticket autostradali (supponendo di classificare i biglietti in 4 gruppi) restituisce l'importo da pagare.

```
1  derived ticketPrice : Tickets -> Natural
2
3  domain Tickets = {1n,2n,3n,4n}
4
5          function ticketPrice(\$n in Tickets) =
6                  switch(\$n)
7                          case (1n) : 3n
8                          case (2n) : 8n
9                          case (3n) : 12n
10                         case (4n) : 17n
11                 endswitch
```

Listing 19: ASM: Funzioni derivate

La funzione viene definita attraverso uno `switch` sull'oggetto che viene passato come parametro.

4.6 Regole

4.6.1 Insert ticket

```
rule r_InsertTicket =
        if (gateStatus = WAITING_TICKET) then
                seq
                        balance := 0n
                        gateStatus := WAITING_PAYMENT
                        toPay := ticketPrice(ticketInserted)
                        message := toPay
                endseq
        endif
```

Listing 20: ASM: Regola insert ticket

Simula l'inserimento del ticket. Inizialmente imposta il balance a 0, si pone poi in status `WAITING_PAYMENT`, dopodichè recupera il costo del pedaggio e infine lo setta come valore di `toPay`. `:=` si utilizza per eseguire l'update di una funzione dinamica.

4.6.2 Insert money

```
rule r_InsertMoney =
        if (gateStatus = WAITING_PAYMENT) then
                seq
                        balance := balance + moneyInserted
                        message := balance
                endseq
        endif
```

Listing 21: ASM: Regola insert money

Questa regola simula l'inserimento di contanti per il pagamento. Incrementa il balance del valore inserito, dopodichè lo stampa a schermo.

4.6.3 Done inserting money

```
rule r_DoneInsertingMoney =
        if (gateStatus = WAITING_PAYMENT) then
                if (balance >= toPay) then
                        par
                                gateStatus := WAITING_CAR_PASS
                                bar := OPENED
                                message := "Thank you!"
                        endpar
                else
                        message := "Please insert more money"
                endif
        endif
```

Listing 22: ASM: Regola done inserting money

Simula la pressione del tasto di completamento del pagamento da parte dell'autista. Essa verifica che l'importo inserito sia sucfficiente per coprire il costo del pedaggio: se questa verifica ha esito negativo, la macchina a stati finiti mostra un messaggio all'utente di richiesta di inserimento di nuovi contanti, altrimenti alza la sbarra e si pone nello stato di `WAITING_CAR_PASS`.

4.6.4 Car passed

```
rule r_CarPassed =
        if (gateStatus = WAITING_CAR_PASS) then
                par
                        toPay := 0n
                        balance := 0n
                        bar := CLOSED
                        gateStatus := WAITING_TICKET
                        message := "Insert ticket"
                endpar
        endif
```

Listing 23: ASM: Regola car passed

Questa regola rappresenta il passaggio dell'automobile davanti alla fotocellula. Essa azzera i contatori `toPay` e `balance`, dopodichè chiude la sbarra e si ripone nello stato iniziale di `WAITING_TICKET`.

4.6.5 Main

```
main rule r_Main =
        seq
                r_InsertTicket[]
                r_InsertMoney[]
                r_DoneInsertingMoney[]
                r_CarPassed[]
        endseq
```

Listing 24: ASM: Regola Main

La regola main è la regola principale da cui viene eseguito il programma. Questa si limita a chiamare le altre regole degli stati. Grazie

alla variabile state, solo una regola per volta potrà essere attiva nella macchina.

4.7 Stato iniziale

```
1 default init initial_state:
2         function toPay = 0n
3         function bar = CLOSED
4         function gateStatus = WAITING_TICKET
5         function message = "Insert ticket"
```

Listing 25: ASM: Stato iniziale

Lo stato iniziale della ASM si occupa di inizializzare la macchina a stati con:

- L'importo da pagare a 0
- Lo stato della barra a `CLOSED`
- Lo stato `WAITING_TICKET`
- Un messaggio iniziale di output