Progetto di

Informatica III (A)

Cyclone, C++, Scala, Abstract State Machines

Samuele Ferri

a.a. 2019-2020

Università degli studi di Bergamo
Scuola di Ingegneria
Corso di laurea in Ingegneria Informatica
v 0.0.1

Indice

1 Cyclone **1**

1.1 Algoritmo RSA . . . 1
1.1.1 Creazione delle chiavi . . . 1
1.1.2 Cifratura del messaggio . . . 2
1.1.3 Decifratura del messaggio . . . 2
1.2 Funzioni . . . 2
1.3 Costrutti Cyclone usati . . . 4
1.3.1 Puntatori * . . . 4
1.3.2 Qualificatore @notnull (puntatore @) . . . 4
1.3.3 Qualificatore @fat (puntatore ?) . . . 4
1.3.4 Qualificatore @zeroterm . . . 5
1.3.5 Qualificatore @numelts(n) (Bounded pointers) . . . 5
1.3.6 Garbage collector e calloc() . . . 5
1.3.7 Files . . . 5

2 C++ **7**

2.1 Classi . . . 7
2.1.1 Ciclista . . . 7
2.1.2 Velocista . . . 8
2.1.3 Passista . . . 10
2.1.4 VelocistaPassista . . . 10
2.1.5 Team . . . 11
2.1.6 Lega . . . 13
2.1.7 Time . . . 14
2.1.8 TemplateStringify.h . . . 15
2.1.9 Main . . . 16

3 Scala **17**

3.1 Classi . . . 17
3.1.1 Persona . . . 18
3.1.2 Ciclista . . . 18
3.1.3 Sponsor . . . 19
3.1.4 Gara . . . 19
3.1.5 Organizzatore . . . 21
3.1.6 Main . . . 23

4 ASM **25**
4.1 Descrizione 25
4.2 Implementazione 25
4.3 Macchina a stati 30
4.4 Scenari 31

1 Cyclone

Cyclone è un dialetto safe del C che permette di evitare i buffer overflow, stringhe non terminate, dangling pointer e altre vulnerabilità endemiche del linguaggio C, senza perdere la potenza e la convenienza della programmazione strutturata.

Cyclone rende sicure le operazioni riguardanti i puntatori, come la gestione di stringhe e l'aritmetica dei puntatori, introducendo i qualificatori dei puntatori che meglio specificano i possibili valori assunti dai puntatori e aggiungono controlli sull'utilizzo degli stessi.

Il progetto sviluppato è un esempio di porting in Cyclone di un algoritmo C usato per la crittografia di stringhe alfanumeriche usando l'algoritmo RSA.

1.1 Algoritmo RSA

RSA è un algoritmo di crittografia asimmetrica a chiave pubblica che permette di cifrare un messaggio attraverso un procedimento che sfrutta le proprietà dei numeri primi.

È basato sull'esistenza di due chiavi: una pubblica conosciuta da tutti usata per cifrare il messaggio e una privata usata solamente per decifrare il messaggio.

Nelle prossime sottosezioni verrà analizzato il funzionamento.

1.1.1 Creazione delle chiavi

Per generare la chiave pubblica e quella privata bisogna seguire i seguenti punti:

1. Scegliere due numeri interi primi p e q
2. Calcolare $n = p \cdot q$ dove n è il modulo della chiave pubblica e di quella privata
3. Calcolare la funzione di Eulero in questo modo: $\Phi(n) = (p-1) \cdot (q-1)$
4. Scegliere un intero e tale che sia coprimo di $\Phi(n)$ (ossia $MCD(e, \Phi(n)) = 1$) e che sia $1 < e < \Phi(n)$; la coppia $\{e, n\}$ sarà la chiave pubblica usata per la cifratura.
5. Scegliere un intero d tale che $d \cdot e \, mod \, \Phi(n) = 1$, ossia pari all'inverso del resto della divisione euclidea tra e ed $\Phi(n)$; la coppia $\{d, n\}$ sarà la chiave privata usata per la decifrazione.

1.1.2 Cifratura del messaggio

Il messaggio in chiaro viene cifrato usando la chiave pubblica $\{e, n\}$ conosciuta da chiunque in questo modo:

$C = M^e (mod\, n)$ dove M il messaggio in chiaro e C è il messaggio cifrato.

Nel codice abbiamo usato T per riferirci ai singoli caratteri da cifrare/decifrare.

1.1.3 Decifratura del messaggio

Il messaggio criptato viene decifrato usando la chiave privata $\{d, n\}$ conosciuta solo dal proprietario in questo modo:

$M = C^d (mod\, n)$ dove M il messaggio in chiaro e C è il messaggio cifrato.

Nel codice abbiamo usato T per riferirci ai singoli caratteri da cifrare/decifrare.

1.2 Funzioni

- gcd

  Descrizione: Controlla se i due numeri interi passati sono o meno coprimi.

  Parametri: *int a, int b*

  Tipo di ritorno: *int*

- inverse

  Descrizione: Calcola l'inverso del resto della divisione euclidea tra a e b.

  Parametri: *int a, int b*

  Tipo di ritorno: *int*

- FindT

  Descrizione: Trova il carattere cifrato/decifrato.

  Parametri: *int a, int m, int n*

  Tipo di ritorno: *int*

- FastExponention

  Descrizione: Algoritmo di moltiplicazione.

  Parametri: *int bit, int n, int *y, int *a*

  Tipo di ritorno: *void*

- KeyGeneration

  Descrizione: Generazioni della chiave privata e pubblica.

  Parametri: -

  Tipo di ritorno: *void*

- Encryption

  Descrizione: Processo di cifratura.

  Parametri: *int value, FILE *out*

  Tipo di ritorno: *void*

- Decryption

  Descrizione: Processo di decifratura.

  Parametri: *int value, FILE *out*

  Tipo di ritorno: *void*

In particolare, la funzione di generazione delle chiavi in C usava dei numeri casuali come partenza, in Cyclone per semplicità sono stati fissati a dei valori predefiniti.

In Cyclone ho usato una stringa preimpostata come messaggio da cifrare in modo da poter evidenziare meglio le caratteristiche di questo dialetto. Per questo motivo è stato necessario aggiungere una funzione di supporto per la clonazione delle stringhe (eseguita in modo safe):

- strclone

  Descrizione: Clonazione delle stringhe.

  Parametri: *const char *@notnull @fat src*

  Tipo di ritorno: *char *@notnull @fat*

1.3 Costrutti Cyclone usati

Vediamo ora in dettaglio quali sono i costrutti di Cyclone usati nel progetto.

1.3.1 Puntatori *

Cyclone introduce delle modifiche riguardo l'uso di normali puntatori * rispetto al linguaggio C:

- Controllo 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)

Cyclone quindi mette a disposizione dei qualificatori per puntatori che meglio specificano gli utilizzi che si possono fare degli stessi.

1.3.2 Qualificatore @notnull (puntatore @)

Il controllo se un puntatore non sia nullo è molto dispendioso; usando questo qualificatore possiamo effettuare il controllo all'assegnamento del valore e evitarlo al suo utilizzo.

Può essere espresso sia con: *** @notnull** che semplicemente con **@.**

Nel progetto è stato usato molto frequentemente, sia per puntatori a stringhe che a files.

1.3.3 Qualificatore @fat (puntatore ?)

Il puntatore definito con questo qualificatore mantiene anche l'informazione sul numero degli elementi dell'array; è possibile accedere a questo numero con **numelts(p)**. Questo qualificatore permette l'aritmetica dei puntatori attraverso il controllo sui limiti dell'array. Tutti gli array possono essere convertiti a @fat (e generalmente viene fatto essendo molto utile).

Questo qualificatore è particolarmente utile per poter conoscere la dimensione delle stringhe senza dover usare strlen (e i problemi legati all'uso del terminatore '\0' generati da questa funzione).

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

Nel progetto è stato usato nella definizione della stringa del messaggio da decifrare e nella funzione di clonazione delle stringhe.

1.3.4 Qualificatore @zeroterm

I puntatori definiti con questo qualificatore indicano che gli array a cui puntano sono terminati da caratteri '\0'. Sono molto utili per la gestione delle stringhe: infatti tutti i puntatori a *char*, fatta eccezione di *char[]* sono di default *@zeroterm*. Esiste anche il qualificatore *@nozeroterm*.

Questo qualificatore permette l'aritmetica dei puntatori (controllando che non vi siano dei terminatori di stringa all'interno), ma questo può diventare dispendioso se non usato in combinazione con @fat.

Nel progetto è stato usato nella definizione della stringa del messaggio da decifrare

1.3.5 Qualificatore @numelts(n) (Bounded pointers)

Indica che il puntatore deve puntare ad un array con esattamente quel numero di elementi. Se l'array contiene più elementi viene generato un warning, se ne contiene di meno un errore.

Nel progetto è stato usato per indicare che gli argomenti della funzione FastExponention sono dei puntatori a un solo elemento *@numelts(1)*.

1.3.6 Garbage collector e calloc()

La funzione *malloc()* di C, non garantisce che la memoria allocata sia inizializzata. La funzione *calloc()* invece azzera tutti i byte della memoria allocata.

Nel progetto è stata sostituita la chiamata *malloc()* con *calloc()*.

Inoltre è possibile affidarsi al garbage collector presente in Cyclone, infatti tutte le variabili allocate nello heap non sono liberate attraverso delle chiamate a *free()* ma gestite in maniera automatica dal garbage collector, che si occuperà di liberare la memoria quando le variabili non sono più referenziate da alcun puntatore.

Nel progetto è stata ugualmente mantenuta la chiamata a *free()*.

1.3.7 Files

Nel progetto sono stati usati i files in modo da poter scrivere il messaggio cifrato in *chiper.txt* e il messaggio successivamente decifrato in *dechiper.txt*. All'apertura di questi file è stato usato il qualificatore *@notnull*.

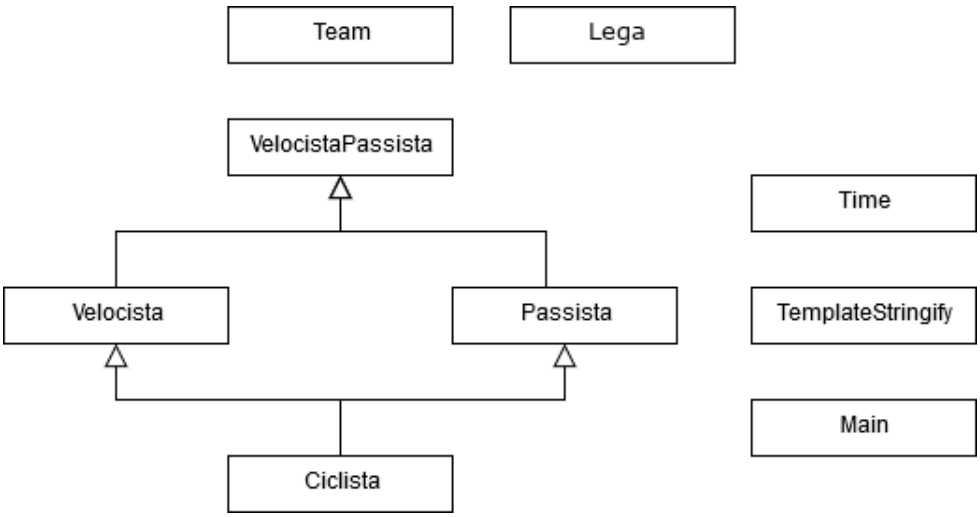
2 C++

Ho sviluppato un'applicazione per mostrare i costrutti tipici del C++: costruttore e distruttore, campi pubblici e privati (normali, statici, costanti, tipi enumerativi) con varie visibilità, metodi virtual e non, overrriding e overloading degli operatori, ereditarietà multipla (diamante), librerie, templates, default arguments, funzioni inline, Standard Template Library (STL) (vector, list, iterator).

Ho scelto il ciclismo come contesto perché si prestava bene per descrivere questi costrutti, in particolar modo all'ereditarietà a diamante.

2.1 Classi

Il progetto è strutturato secondo il paradigma OO, la struttura delle classi è la seguente:



2.1.1 Ciclista

Files: Ciclista.h, Ciclista.cpp

Questa è la classe di base, ha vari campi tra cui una variabile statica *id* usata per creare un identificativo univoco per ogni ciclista; la variabile statica è inizializzata soltanto la prima volta nel file *Ciclista.cpp*.

Nella libreria oltre al costruttore e distruttore sono stati definiti vari metodi virtual e non tra i quali:

```
public:
  Ciclista(string n, string c, string naz, float a,
     ↪ tipociclismo disc); // Costruttore
  virtual ~Ciclista(); // Distruttore (Virtual)
  virtual string toString(); // Stampa (Virtual)
  string getIDString();
  string getNome();
  string getCognome();
  string getNazione();
  string getDisciplina();
  float getEta();
```

Il distruttore è stato definito virtual in modo da consentire di condizionare l'esecuzione del codice secondo il tipo dell'istanza oggetto cui si fa riferimento. Il costruttore, il distruttore e i restanti metodi sono stati ridefiniti nel file *Ciclista.cpp*.

```
Ciclista::Ciclista(string n, string c, string naz, float a,
   ↪ tipociclismo disc) {
  nome = n;
  cognome = c;
  nazionalita = naz;
  eta = a;
  disciplina = disc;
  thisid = ++id; // Incremento l'identificatore univoco
     ↪ progressivo
}

Ciclista::~Ciclista() {
  cout << "Delete Ciclista" << endl;
  free(this);
}
```

Nella libreria è stato usato un enumerativo *tipociclismo* che può assumere i seguenti valori: pista, strada, cross.

```
enum tipociclismo {
  pista, strada, cross
};
```

Viene importato anche il *TemplateStringify.h*, un template per la conversione di interi in stringhe (descritto in seguito).

2.1.2 Velocista

Files: Velocista.h, Velocista.cpp

Velocista è una classe figlia di Ciclista; nella libreria è stata usato un vettore per memorizzare le gare svolte: in particolare è stato usato un *vector* (Standar Template Library (STL)) sulla coppia *pair<KM, Time>* servendosi anche della libreria *Time.h* usata per rappresentare meglio le ore, i minuti e i secondi (descritta in seguito).

```
protected:
  vector<pair<int, Time>> gare;
```

Inoltre viene definita una costante esterna inizializzata nel file *Velocista.cpp* e successivamente usata nel metodo *getIDString().*

```
extern const char* VCONST;
```

Sono stati aggiunti alcuni nuovi metodi nella libreria tra i quali:

```
public:
  void addGara(int km, Time tempo);
  void printGare();
  void printUltimaGara();
```

Nel costruttore presente nel file *Velocista.cpp,* viene invocato anche il costruttore della classe base Ciclista nella member initializer list.

```
Velocista::Velocista(string n, string c, string naz, float a,
   ↪  tipociclismo disc) :
  Ciclista(n, c, naz, a, disc) {
}
```

I nuovi metodi vengono così definiti:

```
void Velocista::addGara(int km, Time tempo) {
  gare.push_back(make_pair(km, tempo));
}

void Velocista::printGare() {
  cout << "\nGare di " << this->getNome() + " " + this->
     ↪ getCognome() << ":" << endl;
  pair<int, Time> p;
  vector<pair<int, Time>>::iterator x;
  for (x = gare.begin(); x != gare.end(); x++) {
    p = *x; // Estraggo
    cout << p.first << " km in " << p.second.getMilitary() <<
       ↪  endl;
  }
}

void Velocista::printUltimaGara() {
  cout << "\nUltima gara di " << this->getNome() + " " + this
     ↪ ->getCognome() << ": " << gare.back().first << " km in
     ↪  "       << gare.back().second.getMilitary() << endl;
     ↪ // Stampa diretta
```

```
}
```

Viene eseguito un inserimento in coda per aggiungere nuove gare disputate dal velocista. Nel ciclo for della stampa delle gare, per scorrere il vettore è stato usato un *iterator*.

Altri metodi come il *toString()* viene fatto l'overriding rispetto al metodo virtual definito nella classe padre.

2.1.3 Passista

Files: Passista.h, Passista.cpp

Passista è una classe figlia di Ciclista; nella libreria è stata aggiunta una variabile protected *numeropodi* e alcuni metodi tra i quali:

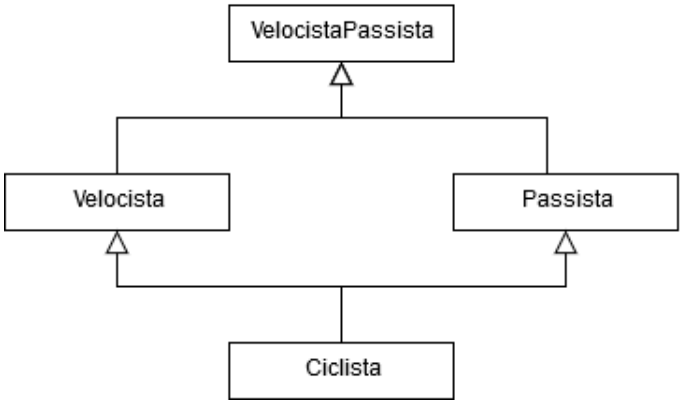
```
protected:
  int numeropodi;

public:
  string getPodi();
  void addPodi(int np);
```

Nel costruttore presente nel file *Passista.cpp,* similmente alla classe Velocista, viene invocato anche il costruttore della classe base Ciclista nella member initializer list. Inoltre vengono anche ridefiniti i metodi della classe padre così come l'overriding del metodo *toString().*

2.1.4 VelocistaPassista

Files: VelocistaPassista.h, VelocistaPassista.cpp

VelocistaPassista ha più classi base che sono velocista e passista; si ha quindi un'ereditarietà multipla, in particolare si viene a creare un'ereditarietà a diamante partendo dalla classe Ciclista.



Nel costruttore vengono invocati entrambi i costruttori delle classi base e di Ciclista nella member initializer list.

```
VelocistaPassista::VelocistaPassista(string n, string c,
   ↪ string naz, float a, tipociclismo disc, int podi) :
    Ciclista(n, c, naz, a, disc), Passista(n, c, naz, a, disc
       ↪ , podi), Velocista( n, c, naz, a, disc) {
}
```

Particolarità di questa classe è che ogni sua istanza potrà usare i metodi di entrambe le classi padre: ad esempio il metodo *printGare()* dalla classe Velocista o il metodo *getPodi()* dalla classe Passista. I metodi in comune come *toString()* sono stati ridefiniti nel file *VelocistaPassista.cpp*.

```
string VelocistaPassista::toString() {
  return "[" + this->Ciclista::getIDString() + "]
     ↪ VelocistaPassista " + nome + " " + cognome + " " +
     ↪ nazionalita + " " + this->getDisciplina() + " " + this
     ↪ ->getPodi() + " podi";
}
```

Il metodo *toString()* è sempre stato ridefinito in ogni classe per permettere una più appropriata stampa della classe di appartenenza. Anche il metodo *getIDString(),* presente in tutte e tre le classi, è stato ridefinito ogni volta in modo tale da apporre le lettere costanti *V, P o VP* davanti al numero identificativo per riconoscere meglio l'istanza dell'oggetto:

```
string VelocistaPassista::getIDString() {
  std::string s = stringify(thisid);
  return VPCONST + s;
}
```

2.1.5 Team

Files: Team.h, Team.cpp

Team è una classe in cui è usata una lista per elencare i membri del team: in particolare è stato usato *list* (Standar Template Library (STL)). Questa lista accetta oggetti di tipo Ciclista, quindi qualsiasi sua classe derivata (Velocista, Passista, VelocistaPassista) va bene.

```
protected:
  std::list<Ciclista*> l;
```

Oltre a questa lista con visibilità protected, ci sono due campi privati e alcuni metodi pubblici in aggiunta al costruttore e al distruttore.

```
private:
  string nomesquadra;
  string origine;

public:
  Team(string ns, string o); // Costruttore
  virtual ~Team(); // Distruttore
  string getNomeSquadra();
  void aggiungi(Ciclista *c);
  void stampa();
  float etaMedia();
  void stampaNaz(string naz);
```

Interessante osservare che nel distruttore, prima di liberare l'istanza della classe, viene anche svuotata la lista dei membri del team:

```
Team::~Team() {
  cout << "Delete Team" << endl;
  l.clear(); // Elimina tutti i giocatori dalla lista
  free(this);
}
```

Nel listato di seguito sono descritti alcuni metodi di questa classe; per scorrere la lista nei vari cicli for è stato usato un *iterator:*

```
float Team::etaMedia() {
  float eta = 0;
  Ciclista *c;
  list<Ciclista*>::iterator x; // Iteratore per scorrere il
     ↪ vettore

  for (x = l.begin(); x != l.end(); x++) {
    c = *x; // Estraggo il ciclista
    eta = eta + c->getEta(); // Estraggo l'età di ogni
       ↪ ciclista del team
  }

  float etamedia = eta / l.size(); // Calcolo l'età media del
     ↪  team
  return etamedia;
}

void Team::stampa() {
  Ciclista *c;
  list<Ciclista*>::iterator x;

  for (x = l.begin(); x != l.end(); x++) {
    c = *x; // Estraggo il ciclista
```

```
    cout << c->toString() << endl; // Stampo ogni ciclista
       ↪ del team
  }
}

void Team::stampaNaz(string naz) {
  Ciclista *c;
  list<Ciclista*>::iterator x;
  cout << "\nCiclisti del team " << this->getNomeSquadra() <<
     ↪  " di origine " + naz + ": " << endl;

  for (x = l.begin(); x != l.end(); x++) {
    c = *x;
    if (c->getNazione() == naz) {
      cout << c->getNome() + " " + c->getCognome() << endl;
    } // Controllo della nazione e stampa del ciclista
  }
}
```

2.1.6 Lega

Files: Lega.h, Lega.cpp

Lega è una classe che racchiude l'unione di più team in cui è usato il *singleton*; infatti esisterà soltanto un'istanza della Lega definita in questo modo:

```
/* TemplateStringify.h  */

class Lega {
public:
  static Lega* getInstance(); // Singleton

  ~Lega();

  string const& getNome() const;
  void setNome(string const &nome);
  void stampa();
  void nuovoTeamIscritto(Team *t);

private:
  static Lega *instance;

  Lega(); // Costruttore privato

  string nome;
```

```
  std::list<Team*> teams;
};
```

Nel file *Lega.cpp* viene definito *getInstance()* per poter avere l'unica istanza della Lega.

```
Lega *Lega::instance = NULL;

Lega::Lega() : nome("") { }

Lega* Lega::getInstance() {
  return instance ? instance : (instance = new Lega()); //
     ↪ Singleton
}
```

2.1.7 Time

File: Time.h

Time è una libreria usata dalle varie classi viste in precedenza per rappresentare meglio il tempo in ore, minuti, secondi. Ad esempio è stata usata nella classe Velocista per memorizzare il tempo di gara nella coppia memorizzata in *vector*.

```
/* Time.h  */

using namespace std;
#include <iostream>
#include "TemplateStringify.h" // Template per la conversione
   ↪  di interi in stringhe

#ifndef TIME_H_
#define TIME_H_

class Time {
private:
  int hour, minutes, seconds;
public:
  Time() {
    hour = minutes = seconds = 0;
  }
  Time(int h) {
    setTime(h, 0, 0);
  }
  Time(int h, int m, int s = 0) { // Default arguments
    setTime(h, m, s);
  }
```

```
  void setTime(int h, int m, int s) {
    hour = (h >= 0 && h < 24) ? h : 0;
    minutes = (m >= 0 && m < 60) ? m : 0;
    seconds = (s >= 0 && s < 60) ? s : 0;
  }

  void printMilitary() {
    cout << hour << ":" << minutes << ":" << seconds;
  }

  string getMilitary() {
    return "" + stringify(hour) + ":" + stringify(minutes) +
       ↪ ":" + stringify(seconds);
  } };

#endif /* TIME_H_ */
```

È presente l'overloading del costruttore, in particolare nel terzo sono stati usati anche i *default arguments*. Sono stati definiti anche alcuni metodi per settare o per stampare il tempo.

2.1.8 TemplateStringify.h

File: TemplateStringify.h

TemplateStringify è un template per la conversione di interi in stringhe.

Dato che in MINGW su Windows la stampa a schermo di interi necessita di un'anteriore conversione usando la libreria sstream, ho creato un template che permette alle altre classi di stampare interi in formato stringa semplicemente chiamando il metodo *stringify*.

```
/* TemplateStringify.h  */

#include <iostream>
#include <string>
#include <sstream> // Metodo str

#ifndef TEMPLATESTRINGIFY_H_
#define TEMPLATESTRINGIFY_H_

template<typename T> inline std::string stringify(const T &t)
   ↪  { // Inline
  std::stringstream string_stream;
  string_stream << t;
  return string_stream.str();
}
```

```
template<typename T, typename ... Args>
inline std::string stringify(const T &first, Args ... args) {
   ↪  // Inline
  return stringify(first) + stringify(args...);
}

#endif /* TEMPLATESTRINGIFY_H_ */
```

Le due funzioni accettano parametri di tipo T-generic e possono convertire in stringhe non soltanto interi; sono funzioni inline in quanto sono definite all'interno di un file *.h*.

2.1.9 Main

File: Main.cpp

Nella classe main ho testato tutte le funzioni dell'applicazione.

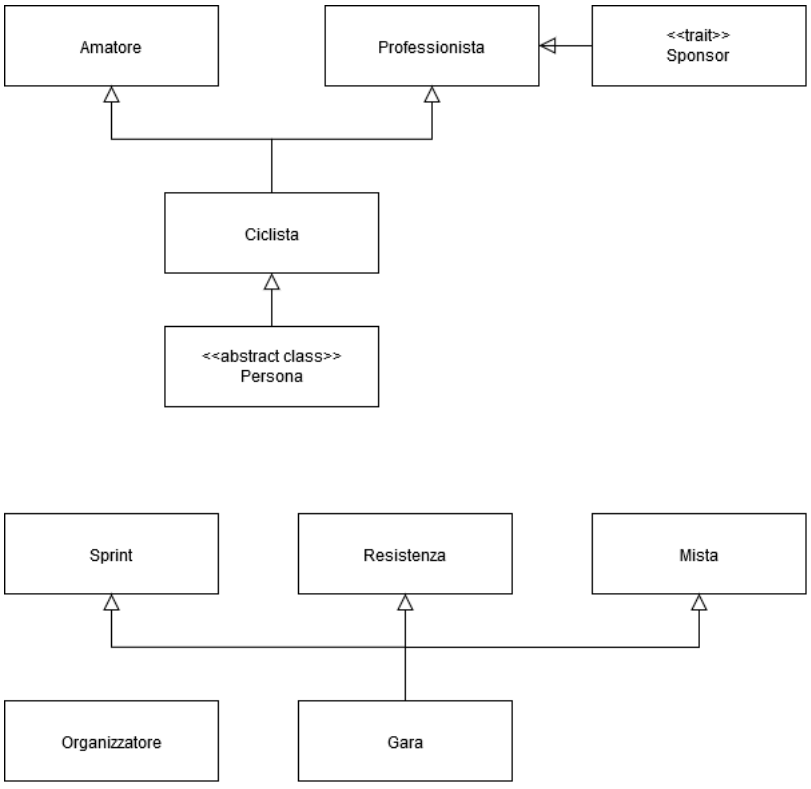
3 Scala

Ho sviluppato un'applicazione per mostrare i costrutti tipici di Scala: classi astratte e non, trait, object, variabili immutabili (val) e mutabili (var), funzioni (def), override di funzioni, default argument, try catch, ereditarità singola delle classi (Scala non permette l'ereditarietà multipla se non attraverso i traits). Inoltre è presente anche la collezione *list* con i metodi *foreach*, *filter* e *map*.

Ho usato ancora il ciclismo come contesto.

3.1 Classi

Il progetto è strutturato secondo il paradigma OO, la struttura delle classi è la seguente:



È presente un'ereditarietà singola sia per Ciclista che per Gara. Inoltre è presente una classe astratta Persona e un *trait* Sponsor.

3.1.1 Persona

File: Persona.scala

Persona è una classe astratta estesa successivamente da Ciclista e definita nel seguente modo:

```
abstract class Persona(val nome: String, val cognome: String,
   ↪  private var _age: Int) {
  if (this._age < 0) throw new IllegalArgumentException

  def age = _age

  def age_=(v: Int): Unit = {
    if (v >= 0) _age = v else throw new
       ↪ IllegalArgumentException
  }

  override def toString: String = { nome + cognome + "(" +
     ↪ age + ")" }
}
```

È stato aggiunto anche un controllo sul valore dell'età con il *try-catch* nel main: nel caso l'età inserita fosse negativa si lancia un'eccezione catturata nel main in questo modo:

```
try {
  val cerror = new Professionista("Luca", "Lucchetti", -22, "
     ↪ Italia", "Conad")
} catch {
  case e: IllegalArgumentException => println("Età non valida
     ↪ ")
}
```

Importante notare anche che alcune variabili sono definite come immutabili *val* e altre mutabili *var*.

3.1.2 Ciclista

File: Ciclista.scala

La classe Ciclista estende la classe Persona con campi e metodi aggiuntivi.

```
// Campi
val nazione: String = naz
var sponsor: String = ""

// Metodi
```

```
def getName() = this.n
def getCognome() = this.c
def getAge() = this.age
def getNazione() = this.naz

override def toString(): String = { nome + " " + cognome + "
   ↪ (" + age + ", " + nazione + ")" }
```

Viene eseguito anche l'override del metodo *toString()*. Nello stesso file ci sono due sottoclassi di Ciclista: Amatore e Professionista definite nel seguente modo:

```
// Sottoclasse Amatore estende Ciclista
class Amatore(n: String, c: String, age: Int, naz: String)
   ↪ extends Ciclista(n: String, c: String, age: Int, naz:
   ↪ String) { }

// Sottoclasse Amatore estende Ciclista con trait Sponsor
class Professionista(n: String, c: String, age: Int, naz:
   ↪ String, s: String = "Nessuno") extends Ciclista(n:
   ↪ String, c: String, age: Int, naz: String) with Sponsor {
  sponsor = s
}
```

La sottoclasse Professionista implementa anche il *trait* Sponsor descritto nella prossima sottosezione; nei parametri della classe Professionista è stato usato anche un default argument sulla stringa *s* per indicare l'assenza di sponsor.

3.1.3 Sponsor

File: Sponsor.scala

Sponsor è un *trait*, ossia un'interfaccia condivisa tra classi simile alle interfacce di Java.

```
// Trait Sponsor
trait Sponsor extends Ciclista {
  def sponsorizzato = println(s"$nome è sponsorizzato da
     ↪ $sponsor")
}
```

3.1.4 Gara

File: Gara.scala

La classe Gara definisce tutte le informazioni relative a una gara svolta come: nome della gara, partecipanti con il relativo tempo, identificativo e tipo di gara.

```
// Campi
val titolo: String = d
var partecipanti: List[(Ciclista, Double)] = a
var tipo: String = ""
var ID: String = ""
```

Come si può notare, per salvare la tupla *(Ciclista, Tempo)* tra i partecipanti a una gara è stata usata una lista.

Inoltre vengono definite tre sottoclassi relative alle varie specialità: Sprint, Resistenza e Mista. Ognuna di queste classi setterà anche l'identificativo e il tipo di gara appropriato.

```
class Sprint(d: String, a: List[(Ciclista, Double)], id: Int)
   ↪  extends Gara(d: String, a: List[(Ciclista, Double)]) {
  if (id < 1000 || id > 9999) println("Errore ID")
  else this.ID = "S" + id
  this.tipo = "Sprint"
}

class Resistenza(d: String, a: List[(Ciclista, Double)], id:
   ↪ Int) extends Gara(d: String, a: List[(Ciclista, Double)
   ↪ ]) {
  if (id < 1000 || id > 9999) println("Errore ID")
  else this.ID = "R" + id
  this.tipo = "Resistenza"
}

class Mista(d: String, a: List[(Ciclista, Double)], id: Int)
   ↪ extends Gara(d: String, a: List[(Ciclista, Double)]) {
  if (id < 1000 || id > 9999) println("Errore ID")
  else this.ID = "M" + id
  this.tipo = "Mista"
}
```

Sempre nella classe Gara vengono definititi i seguenti metodi:

- *printPartecipanti()*: stampa lista di ciclisti partecipanti a una gara.

  ```
  def printPartecipanti() {
    def stampaPartecipanti(a: (Ciclista, Double)) = {
      println(a._1.toString() + " con " + a._2)
      }

      partecipanti.foreach(stampaPartecipanti)
  }
  ```

  È stato usato il *foreach* per scorrere la lista di tutti i partecipanti con il relativo tempo.

- *winner()*: stampa il vincitore di una gara.

```
def winner() {
    var ttt: Double = 9999
    var nome: String = ""
    var cognome: String = ""

    def mintempo(a: (Ciclista, Double)) = {
    if (a._2 < ttt)
          ttt = a._2
    nome = a._1.nome
    cognome = a._1.cognome
  }

    partecipanti.foreach(mintempo)
    println(nome + " " + cognome + " con un tempo di " +
       ↪ ttt)
}
```

È stato usato il *foreach* per scorrere la lista di tutti i partecipanti con il relativo tempo.

- *partecipantiProfessionisti()*: stampa tutti i partecipanti professionisti di una gara.

```
def partecipantiProfessionisti() {
  var ciclistiList: List[Ciclista] = List()

  def doList(a: (Ciclista, Double)) {
    ciclistiList :::= List(a._1)
    }

    partecipanti.foreach(doList)

    ciclistiList
    .filter(_.isInstanceOf[Professionista])
    .map(_.asInstanceOf[Professionista])
    .map(_.sponsorizzato)
}
```

Per poter eseguire il *foreach* ho dovuto prima estrarre solo l'elenco dei ciclisti dalla tupla e poi verificare se siano o meno un'istanza di Professionista attraverso *filter* e *map*; in seguito uso *map* con il metodo *sponsorizzato* (definito nel trait) per stampare lo sponsor dei professionisti.

3.1.5 Organizzatore

File: Organizzatore.scala

Organizzatore è una classe che riceve come parametro una lista di gare e ha i seguenti metodi:

- *insert(g: Gara)*: inserisce una gara in testa alla lista.

```
def insert(g: Gara) = {
  lista :::= List(g)
}
```

- *print()*: stampa tutte le gare gestite.

```
def print() = {
    def stampa(g: Gara) = {
    println("[" + g.ID + " '" + g.titolo + "' " + g.tipo
       ↪ + "]")
    println("Partecipanti:")
    g.printPartecipanti
  }

  lista.foreach(stampa)
}
```

- *find(chiave: String)*: cerca una gara dall'identificativo passato e restituisce un booleano.

```
def find(chiave: String): Boolean = {
    var trovato: Boolean = false

  def trova(g: Gara): Boolean = {
    if (g.ID == chiave) {
      trovato = true; println("Evento trovato: " + g.ID +
         ↪  " '" + g.titolo + "' " + g.tipo)
    }
    trovato
    }
  lista.foreach(trova)
    trovato
}
```

- *vincitoregare()*: stampa i vincitori di ogni gara.

```
def vincitoregare() = {
  def v(g: Gara) = {
    println("Vincitore gara '" + g.titolo + "': ")
    g.winner
    }

    lista.foreach(v)
}
```

3.1.6 Main

File: Main.scala

Nella classe main ho testato tutte le funzioni dell'applicazione.

4 ASM

Ho creato una specifica in ASM relativa a un autolavaggio; il focus è sul menu di selezione e pagamento dell'autolavaggio e sul susseguirsi delle fasi di lavaggio.

È stata usata la *StandardLibrary.asm*.

4.1 Descrizione

Dopo una fase di accensione, il display dell'autolavaggio presenta un menù di selezione delle fasi di lavaggi quali: risciacquo, rulli, cerchi, asciugatura. Il cliente seleziona le fasi di suo interesse, successivamente clicca il pulsante di fine selezione. Si procede al pagamento con il costo totale relativo al numero di fasi scelte. Quando sono stati inseriti un numero sufficiente di soldi per pagare il lavaggio (non dà resto) si procede con le fasi del lavaggio selezionate in ordine. Terminata l'ultima fase, il sistema si riporta sul menù iniziale.

4.2 Implementazione

Vi è un dominio per i soldi *SoldiDomain* (interi tra 0 e 50) e altri due domini relativi agli stati del sistema (*Stato*) e ai tasti selezionati dal cliente dell'autolavaggio (*Selezione*).

Vi sono tre variabili controllate:

- *stato*: indica lo stato attuale del sistema.
- *selezionati*: funzione che memorizza per ogni selezione un valore appartenente al dominio dei soldi; in particolare memorizzerà un 1 se è stata selezionata quella determinata fase o 0 in caso contrario.
- *sommaselezioni*: variabile contenente la somma di tutte le selezioni, usata successivamente per il calcolo del costo totale dell'autolavaggio in base alle fasi selezionate.

Vi sono anche due variabili monitorate:

- *selezione*: permette all'utente di selezionare una determinata fase.
- *soldi*: permette all'utente di inserire una certa quantità di soldi.

```
// DOMAINS
domain SoldiDomain subsetof Integer

enum domain Stato = { START | MENU | PAGAMENTO | LAVORAZIONE
   ↪ | FRISCIACQUO | FRULLI | FCERCHI | FASCIUGATURA }
enum domain Selezione = { RISCIACQUO | RULLI | CERCHI |
   ↪ ASCIUGATURA | FINESELEZIONE }

// FUNCTIONS
dynamic controlled stato : Stato
dynamic controlled selezionati : Selezione -> SoldiDomain
dynamic controlled sommaselezioni : SoldiDomain

monitored selezione : Selezione
monitored soldi : SoldiDomain
```

Lo stato iniziale è:

```
// INITIAL STATE
default init s0: function stato = START
```

Vi sono alcune regole che governano il comportamento della macchina a stati:

Regola **r_START:**

Resetta i valori selezionati e passa allo stato successivo, ossia *MENU*.

```
rule r_START =
  seq
    selezionati(RISCIACQUO) := 0
    selezionati(RULLI) := 0
    selezionati(CERCHI) := 0
    selezionati(ASCIUGATURA) := 0
    stato := MENU
  endseq
```

Regola **r_MENU:**

Questa regola permette di settare a 1 il valore delle fasi selezionate dall'utente. Si passa allo stato *PAGAMENTO* solo quando l'utente clicca *FINESELEZIONE*. Prima di cambiare stato viene anche calcolata la *sommaselezioni* utile in seguito per il calcolo del costo totale del lavaggio.

```
rule r_MENU =
  let ($s = selezione) in
    if ($s != FINESELEZIONE) then
      selezionati($s) := 1
    else
      par
```

```
        sommaselezioni := selezionati(RISCIACQUO) +
           ↪ selezionati(RULLI) + selezionati(CERCHI) +
           ↪ selezionati(ASCIUGATURA)
        stato := PAGAMENTO
      endpar
    endif
  endlet
```

Regola **r_PAGAMENTO:**

In questa regola si attende che l'utente inserisce una quantità di soldi pari o superiore al costo totale dell'autolavaggio calcolato precedentemente. Ricevuto il pagamento si passa allo stato *LAVORAZIONE*.

```
rule r_PAGAMENTO =
  let ($s = soldi) in
    switch (sommaselezioni)
      case 0:
          if ($s >= 0) then
            stato := LAVORAZIONE
          else
            stato := PAGAMENTO
          endif
      case 1:
          if ($s >= 5) then
            stato := LAVORAZIONE
          else
            stato := PAGAMENTO
          endif
      case 2:
          if ($s >= 10) then
            stato := LAVORAZIONE
          else
            stato := PAGAMENTO
          endif
      case 3:
          if ($s >= 15) then
            stato := LAVORAZIONE
          else
            stato := PAGAMENTO
          endif
      case 4:
          if ($s >= 20) then
            stato := LAVORAZIONE
          else
            stato := PAGAMENTO
          endif
    endswitch
```

```
  endlet
```

Regola **r_LAVORAZIONE:**

Questa regola è un centro di smistamento per le varie fasi: infatti si seleziona, in ordine, ogni fase selezionata dall'utente precedentemente. Si passa ai vari stati relativi alle fasi del lavaggio per poi tornare in questa regola di smistamento. Finite tutte le fasi selezionate si ritorna nello stato *START*.

```
rule r_LAVORAZIONE =
  if (selezionati(RISCIACQUO) = 1) then
    stato := FRISCIACQUO
  else
    if (selezionati(RULLI) = 1) then
      stato := FRULLI
    else
      if (selezionati(CERCHI) = 1) then
        stato := FCERCHI
      else
        if (selezionati(ASCIUGATURA) = 1) then
          stato := FASCIUGATURA
        else
          stato := START
        endif
      endif
    endif
  endif
```

Regola **r_FASI:**

In questa regola si esegue la determinata fase e si riporta la specifica variabile *selezionati* a 0. A fine di ogni fase si ritorna nella regola di smistamento *r_LAVORAZIONE*.

```
rule r_FASI =
  if (stato = FRISCIACQUO) then
    par
      // Risciacquo
      selezionati(RISCIACQUO) := 0
      stato := LAVORAZIONE
    endpar
  else
    if (stato = FRULLI) then
      par
        // Rulli
        selezionati(RULLI) := 0
        stato := LAVORAZIONE
      endpar
    else
      if (stato = FCERCHI) then
```

```
        par
          // Cerchi
          selezionati(CERCHI) := 0
          stato := LAVORAZIONE
        endpar
      else
        if (stato = FASCIUGATURA) then
          par
            // Asciugatura
            selezionati(ASCIUGATURA) := 0
            stato := LAVORAZIONE
          endpar
        else
          stato := START // Fine
        endif
      endif
    endif
  endif
```

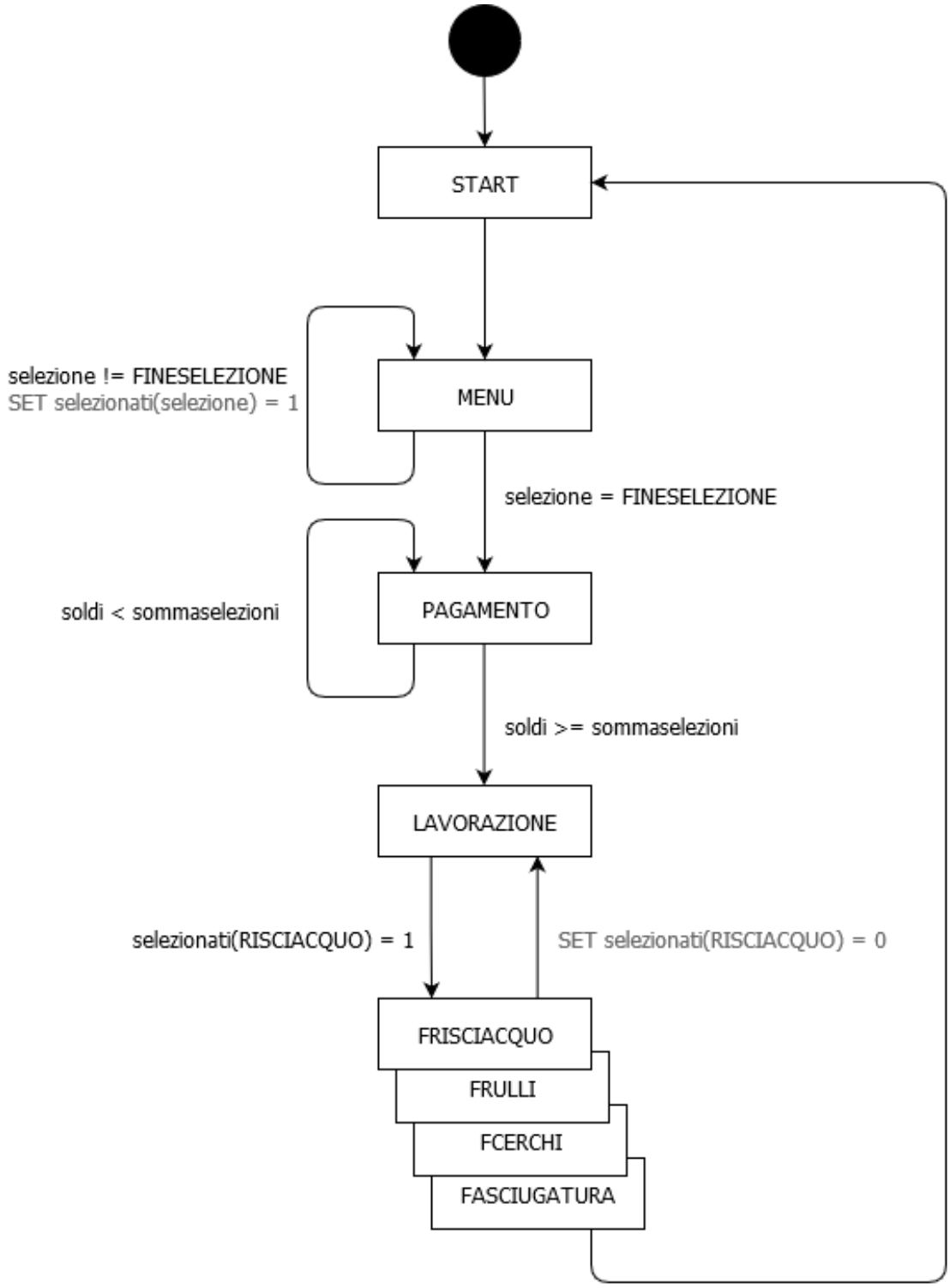
Regola **r_MAIN**:

Questa è la main rule che governa il sistema.

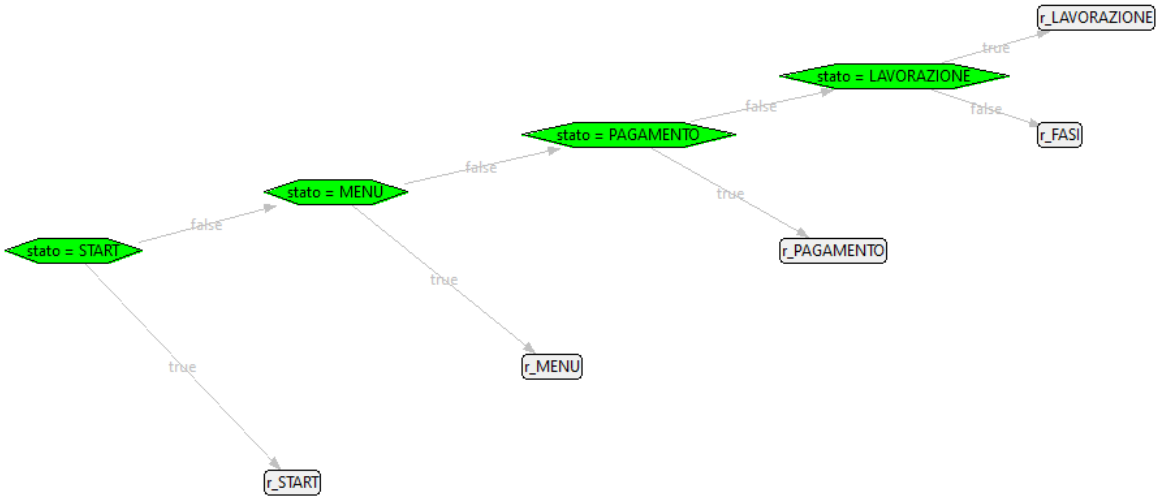
```
main rule r_MAIN =
  if (stato = START) then
    r_START[]
  else
    if (stato = MENU) then
      r_MENU[]
    else
      if (stato = PAGAMENTO) then
        r_PAGAMENTO[]
      else
        if (stato = LAVORAZIONE) then
          r_LAVORAZIONE[]
        else
          r_FASI[]
        endif
      endif
    endif
  endif
```

4.3 Macchina a stati

La macchina a stati in UML è la seguente:



Il modello è il seguente:



4.4 Scenari

Uno scenario testuale con solo la selezione del risciacquo è il seguente:

```
INITIAL STATE:
<UpdateSet - 0>
selezionati(ASCIUGATURA)=0
selezionati(CERCHI)=0
selezionati(RISCIACQUO)=0
selezionati(RULLI)=0
stato=MENU
</UpdateSet>
<State 1 (controlled)>
selezionati(ASCIUGATURA)=0
selezionati(CERCHI)=0
selezionati(RISCIACQUO)=0
selezionati(RULLI)=0
stato=MENU
</State 1 (controlled)>
Insert a symbol of Selezione in [RISCIACQUO, RULLI, CERCHI,
   ↪ ASCIUGATURA, FINESELEZIONE] for selezione:
RISCIACQUO
<State 1 (monitored)>
selezione=RISCIACQUO
</State 1 (monitored)>
<UpdateSet - 1>
selezionati(RISCIACQUO)=1
</UpdateSet>
```

```
<State 2 (controlled)>
selezionati(ASCIUGATURA)=0
selezionati(CERCHI)=0
selezionati(RISCIACQUO)=1
selezionati(RULLI)=0
stato=MENU
</State 2 (controlled)>
Insert a symbol of Selezione in [RISCIACQUO, RULLI, CERCHI,
   ↪ ASCIUGATURA, FINESELEZIONE] for selezione:
FINESELEZIONE
<State 2 (monitored)>
selezione=FINESELEZIONE
</State 2 (monitored)>
<UpdateSet - 2>
sommaselezioni=1
stato=PAGAMENTO
</UpdateSet>
<State 3 (controlled)>
selezionati(ASCIUGATURA)=0
selezionati(CERCHI)=0
selezionati(RISCIACQUO)=1
selezionati(RULLI)=0
sommaselezioni=1
stato=PAGAMENTO
</State 3 (controlled)>
Insert a constant in SoldiDomain of type Integer for soldi:
5
<State 3 (monitored)>
soldi=5
</State 3 (monitored)>
<UpdateSet - 3>
stato=LAVORAZIONE
</UpdateSet>
<State 4 (controlled)>
selezionati(ASCIUGATURA)=0
selezionati(CERCHI)=0
selezionati(RISCIACQUO)=1
selezionati(RULLI)=0
sommaselezioni=1
stato=LAVORAZIONE
</State 4 (controlled)>
<UpdateSet - 4>
stato=FRISCIACQUO
</UpdateSet>
<State 5 (controlled)>
selezionati(ASCIUGATURA)=0
```

```
selezionati(CERCHI)=0
selezionati(RISCIACQUO)=1
selezionati(RULLI)=0
sommaselezioni=1
stato=FRISCIACQUO
</State 5 (controlled)>
<UpdateSet - 5>
selezionati(RISCIACQUO)=0
stato=LAVORAZIONE
</UpdateSet>
<State 6 (controlled)>
selezionati(ASCIUGATURA)=0
selezionati(CERCHI)=0
selezionati(RISCIACQUO)=0
selezionati(RULLI)=0
sommaselezioni=1
stato=LAVORAZIONE
</State 6 (controlled)>
<UpdateSet - 6>
stato=START
</UpdateSet>
<State 7 (controlled)>
selezionati(ASCIUGATURA)=0
selezionati(CERCHI)=0
selezionati(RISCIACQUO)=0
selezionati(RULLI)=0
sommaselezioni=1
stato=START
</State 7 (controlled)>
```