UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

Corso di Laurea Magistrale in Ingegneria Informatica Industriale

Progetti Informatica III A

Pietro Boni

Anno Accademico 2021-2022

Indice

1 Scalvini vs Camuni - Progetto C++

1.1 Funzionamento del programma

Il programma realizzato simula il comportamento di un gioco di combattimento.Vi sono tre tipologie di personaggi selezionabili.

- Scalvini: abitanti della Val di Scalve
- Camuni: abitanti della Valle Camonica
- Scalvinocamuni: un ibrido tra gli abitanti delle valli di cui sopra, creato ai soli fini dell'applicazione

Ogni personaggio giocabile è caratterizzato da un ID univoco che ne permette l'identificazione. Inizialmente, al giocatore verrà richiesta la creazione di due giocatori. Successivamente, la partita si svolge per turni: ad ogni turno al giocatore vengono presentate le seguenti scelte.

- Azione: scelto un personaggio, questo svolge l'azione relativa
- Creazione personaggio: per l'aggiunta di un nuovo personaggio giocabile nella partita
- Combattimento: per far combattere due personaggi giocabili tra di loro

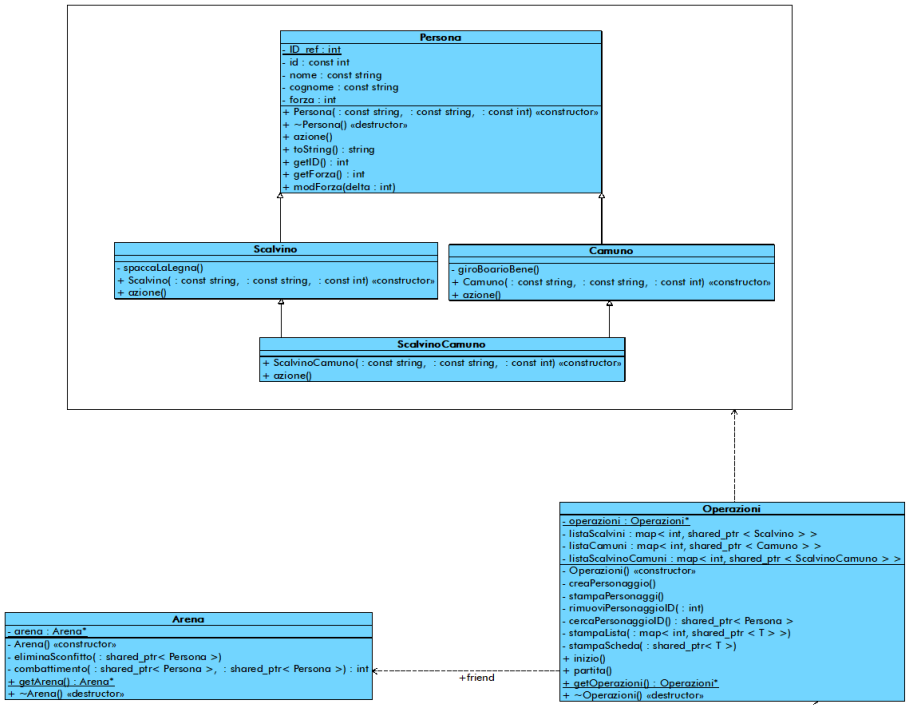
Ogni personaggio è caratterizzato da un certo numero di punti forza: questi possono essere scelti dall'utente o essere assegnati di default in base alla tipologia di personaggio creata. Le azioni di ogni tipologia di personaggio vanno a modificare i relativi punti forza assegnati. In particolare, l'esito di un combattimento è determinato proprio dai punti forza dei due personaggi in gioco.

- Se i due personaggi hanno gli stessi punti forza, il combattimento termina in parità e restano entrambi in gioco
- Altrimenti, il personaggio con più punti forza vince: il personaggio sconfitto viene eliminato dal gioco e i punti forza del vincente aumentano di conseguenza

Durante ogni turno, al giocatore vengono mostrate le liste con le caratteristiche (ID compreso) dei vari personaggi in gioco, divisi per tipologie.

Alla fine di ogni turno al giocatore viene chiesto se vuole continuare la partita o meno. In caso affermativo, inizia un nuovo turno giocabile, altrimenti la partita finisce e di conseguenza anche il programma.

1.2 Abstract class e struttura a diamante



Come mostrato nel class diagram di cui sopra, per evitare il problema della duplicazione dei membri di una classe base in una classe derivata con ereditarietà multipla, è stato adottata una ***struttura a diamante***. Le classi `Scalvino` e `Camuno` ereditano infatti la classe `Persona` come `virtual public`, realizzando il diamante che evita la duplicazione dei membri della classe `Persona` (quindi evitando ambiguità) nella classe derivata `ScalvinoCamuno`, che eredita come `public` le classi `Scalvino` e `Camuno`, tramite il meccanismo dell'ereditarietà multipla. Inoltre, alla base di ogni classe è stata definita la ***classe astratta*** `Personaggio`, con unico metodo da definire `azione()`.

1.3 Uso del qualificatore `friend`

```
16 class Arena {
17     Arena();
18     static Arena *arena;
19     void eliminaSconfitto(shared_ptr<Persona>);
20     int combattimento(shared_ptr<Persona>, shared_ptr<Persona>);
21
22     friend void Operazioni::partita();
```

Come si ppuò notare dal diagramma delle classi e dal codice di cui sopra, la funzione `partita()` della classe `Operazioni` è dichiarata `friend` della classe `Arena`, in modo tale che possa avere accesso ai metodi privati di tale classe. Infatti, la classe `Operazioni` gestisce il funzionamento della partita e dei turni del gioco, ma è indipendente dalla classe `Arena`, che rappresenta l'arena di combattimento tra i personaggi; ossia, non vi è bisogno di ereditarietà in questo caso, da cui l'utilizzo del qualificatore `friend`.

1.4 Overriding, uso di `virtual`, default parameters ed initializer list

Il qualificatore `virtual` è stato utilizzato per ogni distruttore, in particolare per quelli nella struttura a diamante per garantire una corretta eliminazione degli oggetti nella gerarchia. È stato inoltre utilizzato per realizzare l'`overriding` della funzione `azione()` tra le classi nella struttura a diamante, avendo personaggi di tipologia diversa un'azione differente da svolgere, come mostrato di seguito. In particolare, un personaggio della classe `ScalvinoCamuno`, ha come azione lo svolgimento in sequenza delle azioni delle classi direttamente ereditate (quindi `Scalvino` e `Camuno`).

```
14 void Scalvino::spaccaLaLegna() {
15     cout << "_____" << endl << endl;
16     cout << "--- " << toString() << " è andato a spaccare la legna!" << endl;
17     cout << "--- Forza incrementata di 5 punti!" << endl;
18     modForza(5);
19 }
20
21 void Scalvino::azione() { spaccaLaLegna(); }
```

```
15 void ScalvinoCamuno::azione() {
16     Scalvino::azione();
17     Camuno::azione();
18 }
```

Inoltre, per i costruttori delle classi nel diamante, si è preferito l'utilizzo dei ***default parameters*** ad un semplice *overload* per questioni praticità. In particolare, per ogni costruttore si ha un default parameter dato dalla forza del personaggio che se non specificata dall'utente assume, appunto, un valore di default (7 per lo scalvino, 6 per lo scalvinocamuno e 5 per il camuno). Di seguito un esempio. per quanto riguarda l'***initializer list*** questa è stata utilizzata unicamente nel costruttore della classe `Persona`, in quanto altrove sarebbe stata ridondante.

```
30     Persona(const string, const string, const int = 5);
```

```
15 Persona::Persona(const string n, const string c, const int f):
16             nome(n), cognome(c), forza(f) { ++ID_ref; }
```

1.5 Uso del qualificatore `static` e del Singleton pattern

Il qualificatore `static` è stato utilizzato per realizzare il riferimento dal quale assegnare gli ID ai giocatori mano a mano creati. Per ogni giocatore creato, il riferimento viene incrementato di 1 (come si può notare dal costruttore della classe `Persona`, ed il valore ottenuto viene assegnato come ID al personaggio creato.

```
18      static int ID_ref;
19      const int id = ID_ref;
```

Lo `static` è stato anche utilizzato per la realizzazione dei ***Singleton*** delle classi `Operazioni` e `Arena`, per evitare la creazione di diverse istanze non necessarie delle due classi, come mostrato di seguito (in entrambi i casi il costruttore è `private`).

```
15 Arena::Arena() { }
16
17 Arena* Arena::arena = NULL;
18
19 Arena* Arena::getArena() {
20     if(arena == NULL)
21         arena = new Arena();
22     return arena;
23 }
```

```
17 Operazioni::Operazioni() { }
18
19 Operazioni* Operazioni::operazioni = NULL;
20
21 Operazioni* Operazioni::getOperazioni() {
22     if(operazioni == NULL)
23         operazioni = new Operazioni();
24     return operazioni;
25 }
```

1.6 Uso di template, STL (`map` e `iterator`) e qualificatore `auto`

I template sono stati utilizzati nella classe `Operazioni` per generalizzare (e sintetizzare a livello di codice) l'output a schermo di una lista di personaggi di una certa tipologia e la scheda tecnica dei personaggi. In particolare, i metodi generici così realizzati sono generici rispetto alle classi `Scalvino`, `Camuno` e `ScalvinoCamuno`, da cui derivano le tre liste della classe `Operazioni`.

```
34      template<class T> void stampaLista(map<int, shared_ptr<T>>) const;
35      template<class T> void stampaScheda(shared_ptr<T>) const;
```

```
254 //definizioni per il template del metodo stampaLista
255 template void Operazioni::stampaLista<Scalvino>(map<int, shared_ptr<Scalvino>>) const;
256 template void Operazioni::stampaLista<Camuno>(map<int, shared_ptr<Camuno>>) const;
257 template void Operazioni::stampaLista<ScalvinoCamuno>(map<int, shared_ptr<ScalvinoCamuno>>) const;
258
259 //definizioni per il template del metodo stampaScheda
260 template void Operazioni::stampaScheda<Scalvino>(shared_ptr<Scalvino>) const;
261 template void Operazioni::stampaScheda<Camuno>(shared_ptr<Camuno>) const;
262 template void Operazioni::stampaScheda<ScalvinoCamuno>(shared_ptr<ScalvinoCamuno>) const;
```

In particolare, per la realizzazione delle liste di cui sopra si è deciso di utilizzare il container associativo STL `map`, avente come *key* l'ID di un personaggio e come *value* il riferimento al personaggio stesso, in modo tale da facilitare la ricerca di un giocatore quando necessaria.

Inoltre, per scorrere le liste così create, è stato fatto uso degli `iterator` con supporto del qualificatore `auto` per dedurre il tipo dell'iteratore da utilizzare. Gli iteratori così realizzati, permettono nei metodi della classe `Operazioni` di scorrere le liste di riferimento per svolgere diversi compiti, come ad esempio stampare in output le liste, cercare un personaggio tramite ID per eliminarlo dalla lista o per selezionarlo in quanto scelto dal giocatore.

```
29      map<int, shared_ptr<Scalvino>> listaScalvini;
30      map<int, shared_ptr<Camuno>> listaCamuni;
31      map<int, shared_ptr<ScalvinoCamuno>> listaScalvinoCamuni;
```

```
142      //stampa l'intera lista
143      for(auto i = lista.begin(); i != lista.end(); ++i)
144          stampaScheda(i->second);
```

```
236     //cerca nella lista un personaggio in base all'ID
237     for(auto i = listaScalvini.begin(); i != listaScalvini.end(); ++i)
238         if(i->first == id)
239             return i->second;
```

1.7 Uso degli smart pointers

Gli ***smart pointer*** sono stati impiegati al posto dei puntatori raw per una più efficiente gestione della deallocazione della memoria, visto il potenziale numero elevato di allocazioni e deallocazioni necessarie. In particolare, come visto sopra, per la creazione delle liste relative alle tre tipologie di personaggio si è utilizzato lo `shared_pointer`, in quanto durante un turno della partita si potrebbero avere più puntatori che puntano allo stesso personaggio allocato sullo heap.

Per quanto riguarda invece i *Singleton* relativi alle classi `Arena` e `Operazioni`, si è deciso di utilizzare lo `unique_pointer`, in quanto queste vengono istanziate una sola volta nel corso del programma (`Operazioni` per dare il via alla partita e `Arena` all'inizio della partita), per cui non avranno sicuramente altri puntatori alla loro istanza.

```
150 //gestisce le azioni di una partita
151 void Operazioni::partita(){
152
153     //singola istanza di Arena per la partita
154     unique_ptr<Arena> arena(Arena::getArena());
```

```
19      //gestisce la partita
20      unique_ptr<Operazioni> operazioni(Operazioni::getOperazioni());
```

```
71      //aggiunta personaggio nella lista specifica in base alla scelta fatta
72      switch(scelta){
73      //SCALVINO
74      case 's':
75      case 'S':
76      {
77          shared_ptr<Scalvino> s;
78          //differenzio il costruttore in base alla scelta fatta sulla forza del personaggio
79          if(forza>0)
80              s = shared_ptr<Scalvino>(new Scalvino(nome, cognome, forza));
81          else
82              s = shared_ptr<Scalvino>(new Scalvino(nome, cognome));
83          //accoppio lo scalvino creato (value) al suo ID (key) nella lista
84          listaScalvini.insert(make_pair(s->getID(), s));
85          break;
86      }
```

Gli smart pointer così realizzati permettono una gestione efficace della memoria in fase di chiusura della partita: quando il giocatore decide di terminare la partita, l'istanza di `Operazioni` viene deallocata automaticamente dallo `unique_pointer` associato e così di conseguenza anche l'istanza di `Arena` e a catena tutte le istanze ancora attive dei personaggi che si trovavano ancora nelle liste a fine partita.

2 Combattimenti Fantasy - Progetto Scala

2.1 Funzionamento del programma

Il programma realizzato, sulla falsa riga di quello proposto in C++, simula il combattimento tra diverse classi di personaggi provenienti da un mondo fantasy. Le classi di personaggi utilizzabili sono le seguenti.

- Umani, caratterizzati da un valore medio di punti forza. Tale classe si specializza a sua volta in due sottoclassi
  - Guerrieri, caratterizzati da un valore di punti forza superiore rispetto ad un umano comune
  - Stregoni, caratterizzato da un basso valore di punti forza, bilanciato però dalla specialità della classe
- Orchi, caratterizzati da un alto valore di punti forza
- Elfi, caratterizzati dal più alto valore di punti forza

In particolare, per ogni classe il valore dei punti forza può essere specificato a piacere o lasciato al valore di default.

Il funzionamento del programma è simile a quello precedente, con la differenza che non è stata implementata la possibilità di input da parte dell'utente (viene meno la parte interattiva). In particolare, ad ogni "turno" i personaggi possono combattere tra loro oppure dare sfoggio della propria specialità, specifica per ogni classe, per aumentare i proprio punti forza in una certa misura a seconda della classe di appartenenza. Come nel programma precedente, i personaggi giocabili sono raccolti in una lista e reperibili tramite un ID univoco, sia per la fase di combattimento che per l'eliminazione dalla lista dei personaggi giocabili (causa sconfitta o decisione arbitraria).

2.2 Abstract class, uso di getter/setter e `trait`

```
//classe astratta per la definizione di un personaggio
abstract class Personaggio extends Specialita{

  //ID univoco del personaggio
  private val _ID = Personaggio.getID
  //nickname personaggio
  private val _nick = ""
  //classe del personaggio
  private val _classe = ""
  //forza personaggio
  private var _forza = 0
```

La ***classe astratta*** `Personaggio` è stata utilizzata come classe base per tutte le classi principali di personaggi utilizzabili. In essa sono dichiarati i membri comuni a tutte le classi (`ID`, `nickname`, `classe` e `forza`) oltre che i ***getter*** ed il ***setter*** del solo membro `forza`, in quanto l'unico modificabile (dichiarato tramite qualificatore `var`, gli altri tramite `val`). Sia getter che setter sono stati dichiarati secondo la convenzione del linguaggio utilizzato. Vi è inoltre il metodo comune `printP` per visualizzare in modo compatto tutte le informazioni relative ad un personaggio.

```
//getters
def ID = _ID
def nick = _nick
def classe = _classe
def forza = _forza

//setter forza
def forza_= (newForza: Int) = _forza += newForza
```

La classe `Personaggio`, inoltre, estende anche il `trait Specialita` (creato a solo scopo didattico), che dichiara il metodo `specialita`, utilizzato dalle varie classi di personaggi per definire la propria particolare specialità.

```
//trait per la specialità di un personaggio
trait Specialita {

  def specialita

}
```

2.3 Uso di Companion Objects, overload e default parameters

I ***Companion Objects*** sono stati qui utilizzati con due obiettivi principali.

- Permettere l'istanziazione di un personaggio senza l'utilizzo del qualificatore `new`
- Permettere l'assegnazione ad ogni nuovo personaggio creato di un ID univoco per una corretta indicizzazione

La prima funzionalità è stata implementata implementando un companion object per ogni sottoclasse derivata dalla classe astratta `Personaggio`. In questo modo,

l'istanziamento dei singoli personaggi risulta sia più conciso che più comodo. In tali companion object è stato inoltre realizzato il meccanismo dell'***overload***, per garantire la creazione di un personaggio specificando o meno il valore dei punti forza.

```
//companion per l'istanziamento di uno stregone
object Stregone{

  //nel caso siano specificati sia nick che forza
  def apply(nick: String, forza: Int): Stregone = {
    var m = new Stregone(nick, forza);
    m
  }

  //nel caso sia specificato solo il nick
  def apply(nick: String): Stregone = {
    var m = new Stregone(nick);
    m
  }
}
```

Infatti, ad ogni classe è associato un costruttore in cui il parametro `forza`, se non specificato, è assegnato di default ad un valore specifico per ogni classe.

```
class Orco (override val nick: String, override var forza: Int = 15) extends Personaggio
```

```
class Elfo (override val nick: String, override var forza: Int = 20) extends Personaggio
```

La seconda funzionalità per cui è stato utilizzato il meccanismo del companion object è stata implementata relativamente alla classe `Personaggio`: per ogni personaggio creato, infatti, si vuole l'assegnazione di un ID univoco. Non disponendo di un qualificatore `static` si è appunto fatto uso di un companion object, nel quale si ha l'inizializzazione di un ID di riferimento a 0, il quale viene incrementato di 1 alla creazione di ogni personaggio, per l'assegnazione dello stesso al personaggio creato.

```
object Personaggio{
  //riferimento per assegnare un ID univoco ai personaggi
  var ID_ref = 0

  //incrementa il riferimento per l'ID e lo restituisce per l'assegnamento al personaggio
  def getID: Int = {
    ID_ref += 1
    ID_ref
  }
}
```

2.4 Uso di ereditarietà ed overriding

Come detto sopra, tutte le classe principali sono sottoclassi della classe astratta `Personaggio`. Inoltre, la classe `Umano` si specializza a sua volta nelle due sottoclassi `Guerriero` e `Stregone`. Inoltre, tutte le classi giocabili specializzano tramite ***overriding*** la funzione `specialita` "ereditata" dal `trait Specialita`: la specializzazione consiste semplicemente nella chiamata di una funzione privata specifica per classe che permette un aumento della forza del personaggio di un certo numero di punti.

```
//specialità dello stregone
private def allenamento = {
  println("--- " + nick + " usa una stregoneria! Forza incrementata di 10!")
  println
  forza += 10
}

//chiama la specialità del personaggio
override def specialita = allenamento
```

2.5 Uso di Object e `LinkedHashMap`

Similmente al programma precedente, sono state realizzati l'`Arena` ed un `Gestore` come singleton, sfruttando il meccanismo della classe `Object`, che assolve appunto al ruolo di singleton. Sempre come in precedenza, `Arena` si occupa della gestione dei combattimenti tra personaggi, mentre `Gestore` della totalità dei personaggi giocabili. Qui la lista dei personaggi giocabili è stata realizzata con l'utilizzo di `LinkedHashMap`, classe che permette la creazione di una lista caratterizzata da coppie (`ID`, `Personaggio`), ordinate in base a valori crescenti dell'ID. Si è optato per questa soluzione per comodità e per sfruttare al meglio le classi standard del linguaggio Scala. Di seguito alcuni esempi di utilizzo.

```
//gestisce i personaggi in gioco tramite una lista
object Gestore {

  //lista dei personaggi
  private var listaPersonaggi: LinkedHashMap[Int, Personaggio] = LinkedHashMap()

  //aggiunge un personaggio a listaPersonaggi
  def addPersonaggio(p: Personaggio) = listaPersonaggi.put(p.ID, p)

  //aggiunge una lista di personaggi a listaPersonaggi
  def addListaPersonaggi(lista: List[Personaggio]) = lista.foreach(p => addPersonaggio(p))

  //restituisce il personaggio corrispondente all'ID passato
  def getPersonaggio(id: Int): Personaggio = listaPersonaggi.getOrElse(id, null)
```

3 Slot Machine - Progetto ASM

3.1 Funzionamento della slot machine

Si è deciso di implementare con ASM una slot machine virtuale con regole personalizzate. La slot machine è composta da tre rulli con 13 slot per ospitare i simboli utilizzati. I simboli utilizzati sono 8, divisi in due categorie:

- frutti: uva, mela, pera, ribes, kiwi
- speciali: SPIN, WILD, JACKPOT

Inizialmente il giocatore parte con 100 € nel portafoglio, mentre la cassa della slot machine con 300 €. Per ogni spin (turno) il giocatore seleziona una puntata fra quelle disponibili. A questo punto, i rulli della slot girano casualmente fino a fermarsi, riportando sulla riga evidenziata all'utente la combinazione di simboli ottenuta.

In base alla combinazione ottenuta si ha un comportamento diverso della slot machine:

- TRIS di JACKPOT: il giocatore vince un importo pari a 3000 €
- TRIS di WILD: il giocatore vince un importo pari a 300 €
- TRIS di frutti o SPIN: il giocatore vince un importo pari alla puntata scommessa x20
- WILD + doppio JACKPOT o doppio WILD + JACKPOT: il giocatore vince un importo pari alla puntata scommessa x50
- frutto o SPIN + doppio WILD: il giocatore vince un importo pari a 150 €
- WILD + doppio frutto (uguale) o doppio SPIN: il giocatore vince un importo pari alla puntata scommessa x15

Una regola particolare riguarda il simbolo SPIN: se ottenuto, in aggiunta alle eventuali vicnite del turno, tale simbolo garantisce uno spin gratuito al giocatore, settato alla puntata massima, quindi 20 €. Il numero di spin gratuiti è cumulabile in base al numero di SPIN ottenuti, anche in turni differenti. Inoltre, non è possibile durante la partita aggiungere soldi al portafoglio o alla cassa della slot machine.

La partita del giocatore termina nei seguenti casi:

- sono stati giocati 30 spin
- il giocatore non ha più soldi nel portafoglio o è in negativo (quindi deve dei soldi al banco)
- la cassa della slot machine è vuota o è in negativo (quindi il banco deve dei soldi al giocatore)

La partita risulta vinta se il giocatore ha vinto più soldi rispetto a quelli spesi, viceversa è persa (nulla di fatto in caso non ci sia stato né guadagno né perdita).

3.2 Dichiarazioni - Uso di enum, subsetof e Prod per domini custom, variabili controlled, monitored e 'shared'

```
signature:

    //simboli sui rulli
    enum domain Simbolo = {UVA | MELA | PERA | RIBES | KIWI | SPIN | WILD | JACKPOT}

    domain Puntata subsetof Integer     //dominio per i valori delle puntate
    domain SlotRullo subsetof Integer   //dominio per il numero di slot di un rullo
    domain Spin subsetof Integer        //dominio per il numero di spin (turni) disponibili
    domain SimboloAss subsetof Prod(SlotRullo, Simbolo) //Simbolo associato ad uno slot del rullo

    //rappresentano la fila orizzontale dei simboli ottenuti dallo spin dei tre rulli
    controlled rullo1: Simbolo
    controlled rullo2: Simbolo
    controlled rullo3: Simbolo

    controlled cassaSlot: Integer       //soldi nella cassa della slot
    controlled portafoglio: Integer     //soldi nel portafoglio del giocatore
    controlled guadagno: Integer        //se positivo il giocatore vince, se negativo perde
                                        //se nullo -> nulla di fatto
    controlled risultato: String        //risultato della partita
    controlled spin: Spin               //numero di spin giocati
    controlled spinOmaggio: Natural     //spin omaggio alla massima puntata (uscito spin)

    monitored puntata: Puntata          //puntata fatta dal giocatore
    controlled puntataS: Puntata        //valore della puntata controllato dalla slot

    static getRisultato: String         //per visualizzare il risultato della partita
```

Per quanto riguarda le variabili, l'unica sotto diretto controllo dell'utente è la variabile `puntata`, in quanto è l'unica scelta che l'utente può fare durante il gioco. In particolare, la puntata dello spin attuale è gestita anche dalla variabile `puntataS`, di topo `controlled`: una volta che l'utente ha scelto una puntata, il valore di quest'ultima viene assegnato a `puntataS` ed eventuali vincite gestite attraverso tale variabile. È stata scelta una soluzione di questo tipo per simulare il funzionamento di una variabile di tipo `shared`, non ancora supportata dal simulatore. Infatti, la puntata deve poter essere gestita anche dalla slot nel caso in cui si ricevano spin omaggio (in modo tale da settare la puntata al valore massimo possibile, senza scelta da parte del giocatore).

3.3 Definizioni: Update Rule, Block Rule, Sequence Rule, Conditional Rule e Choose Rule

```
domain Puntata = {1, 2, 5, 10, 20}
domain SlotRullo = {1:13}
domain Spin = {0:30}
domain SimboloAss = { (1, UVA), (2, MELA), (3, PERA), (4, RIBES), (5, KIWI),
                      (6, UVA), (7, MELA), (8, PERA), (9, RIBES), (10, KIWI),
                      (11, SPIN), (12, WILD), (13, JACKPOT) }
```

I domini così definiti specificano le seguenti caratteristiche della slot machine:

- `domain Puntata` → valori disponibili per le puntate del giocatore (1 €, 2 €, 5 €, 10 € e 20 €)
- `domain SlotRullo` → ogni rullo è composto da 13 slot, ognuno dei quali andrà ad ospitare un simbolo
- `domain Spin` → si ha un massimo di 30 spin - ossia di turni - per partita
- `domain SimboloAss` → specifica le corrispondenze tra gli slot di un rullo e i simboli ammissibili (specificati nel `enum domain Simbolo`), sottoforma di associazione tra il numero dello slot e nome corrispondente del simbolo (praticamente è una `Map`)

```
main rule r_Main =

    if(spin=30 or portafoglio<=0 or cassaSlot<=0) then
        risultato := getRisultato
    else
        seq
            spin := spin + 1

            if spinOmaggio > 0n then
                seq
                    spinOmaggio := iton(spinOmaggio - 1n)
                    puntataS := 20
                endseq
            else
                seq
                    r_updateVincita[-puntata]
                    puntataS := puntata
                endseq
            endif

            choose $sa1 in SimboloAss, $sa2 in SimboloAss, $sa3 in SimboloAss with true do
                par
                    rullo1 := second($sa1)
                    rullo2 := second($sa2)
                    rullo3 := second($sa3)
                endpar

            if rullo1 = rullo2 and rullo2 = rullo3 then
                r_tris[rullo1]
            else
                if rullo1 = WILD or rullo2 = WILD or rullo3 = WILD then
                    r_checkSpecial[WILD]
                endif
            endif

            if rullo1 = SPIN or rullo2 = SPIN or rullo3 = SPIN then
                r_checkSpecial[SPIN]
            endif
        endseq
    endif
```

In `main rule r_Main` sono specificati i comportamenti desiderati da parte della slot machine durante uno spin. Inizialmente si verifica se la partita del giocatore è finita: in questo caso viene visualizzato il risultato ottenuto dal giocatore, mentre in caso contrario si procede ad un nuovo spin. Si aggiornano poi i valori di `portafoglio`, `guadagno` e `cassaSlot` in base alla puntata scelta; nel caso il giocatore abbia degli spin omaggio non vengono aggiornati i valori monetari, in quanto la puntata viene settata automaticamente alla massima possibile (come dei soldi virtuali 'regalati' al giocatore).

Si procede poi allo spin dei rulli della slot machine, per cui per ogni rullo viene selezionato un simbolo casuale fra i 13 disponibili per ogni rullo. In base alle combinazioni di simboli ottenuti, se queste riguardano una possibile vincita da parte del giocatore si viene rimandati alla funzione di gestione corrispondente, altrimenti il turno finisce.

```
//verifica se uno dei simboli è uno spin o uno wild
macro rule r_checkSpecial($sp in Simbolo) =
    if rullo1 = $sp then
        switch($sp)
            case WILD:
                r_wild[rullo2, rullo3]
            case SPIN:
                r_spin[rullo2, rullo3]
        endswitch
    else
        if rullo2 = $sp then
            switch($sp)
                case WILD:
                    r_wild[rullo1, rullo3]
                case SPIN:
                    r_spin[rullo1, rullo3]
            endswitch
        else
            if rullo3 = $sp then
                switch($sp)
                    case WILD:
                        r_wild[rullo1, rullo2]
                    case SPIN:
                        spinOmaggio := spinOmaggio + 1n //unico spin
                endswitch
            endif
        endif
    endif
```

la regola `r_checkSpecial` gestisce i casi in cui almeno uno dei simboli ottenuti dallo spin sia un simbolo speciale tra SPIN e WILD. Nel caso in cui anche un solo rullo corrisponda ad uno di questi simboli, si viene rimandati alla regola di gestione

corrispondente, tranne nel caso in cui il terzo rullo sia uguale ad uno SPIN: in questo caso, infatti, non c'è nessuna combinazione vincente ulteriore possibile, per cui al giocatore viene assegnato uno spin omaggio gratis e il turno finisce.

```
//gestione doppio wild + simbolo
macro rule r_doppioWild($r in Simbolo) =
    switch($r)
        case JACKPOT:
            r_updateVincita[puntataS*50]        //tris doppio wild + jackpot -> puntata*50
        otherwise
            r_updateVincita[150]                //tris doppio wild + frutto -> 150 €
    endswitch


//gestione wild + doppio simbolo
macro rule r_wildDoppioSimbolo($r in Simbolo) =
    switch($r)
        case JACKPOT:
            r_updateVincita[puntataS*50]        //tris wild + doppio jackpot -> puntata*50
        otherwise
            r_updateVincita[puntataS*15]        //tris wild + doppio frutto -> puntata*15
    endswitch


//verifica di altri wild eventuali
macro rule r_wild($r1 in Simbolo, $r2 in Simbolo) =
    //wild con doppio simbolo
    if $r1=$r2 then
        r_wildDoppioSimbolo[$r1]
    else
        //doppio wild
        if $r1=WILD then
            r_doppioWild[$r2]
        else
            if $r2=WILD then
                r_doppioWild[$r1]
            endif
        endif
    endif
```

Le regole che gestiscono i casi in cui almeno uno dei simboli dello spin corrente sia un WILD sono `r_wild`, `r_wildDoppioSimbolo` e `r_doppioWild`.

- `r_wild` → è la prima regola di verifica chiamata nel caso uno dei rulli abbia come simbolo un WILD. Prende i riferimenti agli altri due rulli per verificare eventuali combinazioni di vincita per il giocatore, rimandando eventualmente alle relative funzioni di gestione
- `r_wildDoppioSimbolo` → chiamata nel caso in cui un rullo ha come simbolo WILD e gli altri due lo stesso simbolo (diverso da WILD). Prende il riferimento ad uno dei due rulli non WILD e chiama la regola per l'aggiornamento dei valori monetari passando il valore della puntata:
  - x15 nel caso in cui il simbolo sui rulli non WILD non sia un JACKPOT
  - x50 nel caso in cui il simbolo sui rulli non WILD sia un JACKPOT

- `r_doppioWild` → chiamata nel caso in cui due rulli abbiano assunto come valore WILD. Prende il riferimento al rullo non valutato e chiama la regola per l'aggiornamento dei valori monetari passando:
  - il valore della puntata x50 nel caso in cui il rullo abbia come simbolo JACKPOT
  - 150 (€) altrimenti

```
//gestione spin
macro rule r_spin($r1 in Simbolo, $r2 in Simbolo) =
    if $r1 = $r2 and $r1 = SPIN then        //tris di spin
        spinOmaggio := spinOmaggio + 3n
    else
        if $r1 = SPIN or $r2 = SPIN then    //coppia di spin
            spinOmaggio := spinOmaggio + 2n
        else
            spinOmaggio := spinOmaggio + 1n //unico spin
        endif
    endif
```

Gli SPIN sono gestiti dalla regola `r_spin`. A seconda del numero di SPIN ottenuti, viene conseguentemente aggiornato il numero di spin omaggio del giocatore.

```
//gestione tris
macro rule r_tris($r in Simbolo) =
    switch($r)
        case JACKPOT:
            r_updateVincita[3000]           //tris jackpot -> 3000€
        case WILD:
            r_updateVincita[300]            //tris wildcard -> 300€
        otherwise
            r_updateVincita[puntataS*20]    //tris figure -> puntata*20
    endswitch
```

I tris sono gestiti dalla regola `r_tris`, che chiama la regola per l'aggiornamento dei valori monetari in base al tipo di tris ottenuto.

- Il giocatore vince 3000 € nel caso di tris di JACKPOT
- Il giocatore vince 300 € nel caso di tris di WILD
- Il giocatore vince un valore dato dalla puntata fatta x20 nel caso di tris con un frutto o uno SPIN

```
//aggiorna i saldi in caso di vittoria (o puntata) del giocatore
macro rule r_updateVincita ($win in Integer) =
    par
        cassaSlot := cassaSlot - $win
        guadagno := guadagno + $win
        portafoglio := portafoglio + $win
    endpar
```

La regola `r_updateVincita` gestisce i valori monetari sotto controllo, aggiornandoli in base alla vincita del turno del giocatore. Oltre a ciò, aggiorna i valori all'inizio di ogni turno in base al valore della puntata del giocatore (a meno di spin gratuiti).

```
//visualizzazione risultato di fine partita
function getRisultato =
    if guadagno > 0 then
        "Hai sbancato!"
    else
        if guadagno < 0 then
            "Sara' per un'altra volta!"
        else
            "Ci hai almeno provato?"
        endif
    endif
```

La funzione statica `getRisultato` permette la visualizzazione in output di una frase personalizzata in base alla vittoria o meno del giocatore, determinata in base al valore del guadagno della partita.

3.4 Valori iniziali

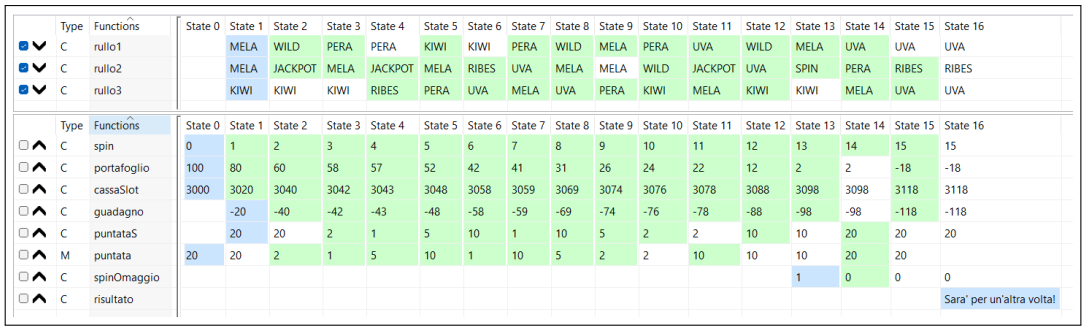
```
default init s0:

    function portafoglio = 100  //il giocatore parte con 300€
    function cassaSlot = 3000   //la slot parte da 3000€ (massima somma in vincita)
    function guadagno = 0
    function spin = 0
    function spinOmaggio = 0n
```

La cassa iniziale della slot parte da un valore di 3000 € e il giocatore parte con 100 €; inoltre, sia al portafoglio che alla cassa della slot non possono essere aggiunti soldi nel corso della partita. Il guadagno iniziale, così come lo spin corrente e gli spin omaggio (dati dall'estrazione di uno SPIN) sono impostati a 0.

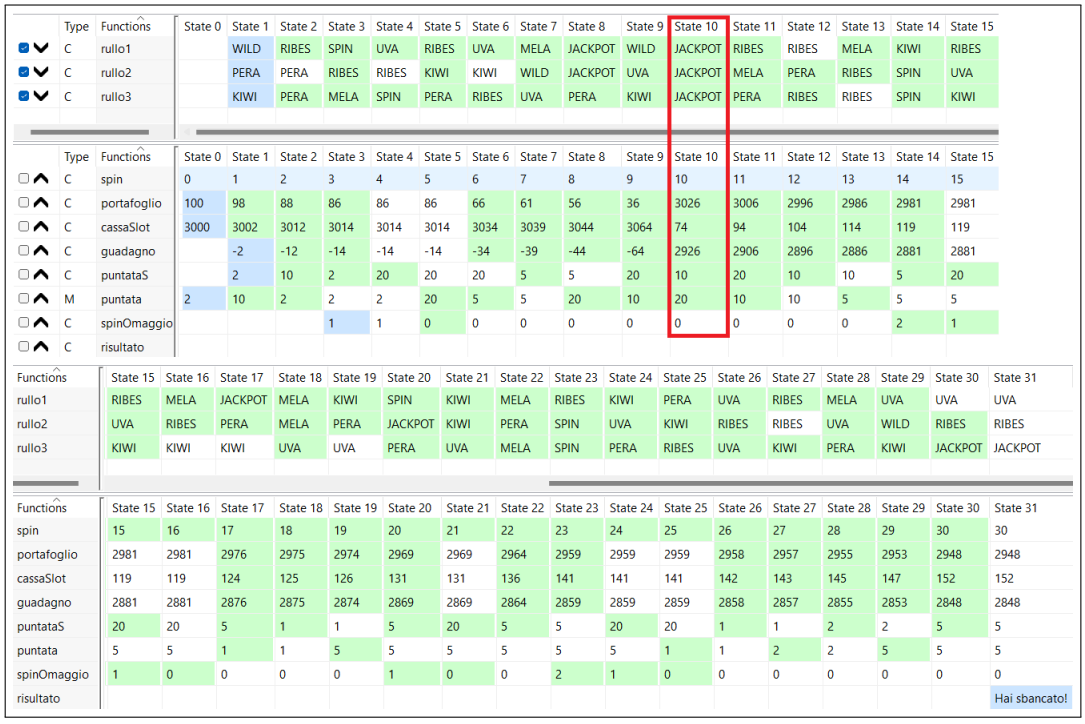
3.5 Simulazioni

Di seguito alcune simulazioni del comportamento della slot machine in diverse partite. Non si è potuto procedere alla verifica tramite `Avalla` a causa dell'incertezza dovuta all'estrazione dei simboli dei rulli per ogni turno.



Type	Functions	State 0	State 1	State 2	State 3	State 4	State 5	State 6	State 7	State 8	State 9	State 10	State 11	State 12	State 13	State 14	State 15	State 16
C	rullo1		MELA	WILD	PERA	PERA	KIWI	KIWI	PERA	WILD	MELA	PERA	UVA	WILD	MELA	UVA	UVA	UVA
C	rullo2		MELA	JACKPOT	MELA	JACKPOT	MELA	RIBES	UVA	MELA	MELA	WILD	JACKPOT	UVA	SPIN	PERA	RIBES	RIBES
C	rullo3		KIWI	KIWI	KIWI	RIBES	PERA	UVA	MELA	UVA	PERA	KIWI	MELA	KIWI	KIWI	MELA	UVA	UVA

Type	Functions	State 0	State 1	State 2	State 3	State 4	State 5	State 6	State 7	State 8	State 9	State 10	State 11	State 12	State 13	State 14	State 15	State 16
C	spin	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	15
C	portafoglio	100	80	60	58	57	52	42	41	31	26	24	22	12	2	2	-18	-18
C	cassaSlot	3000	3020	3040	3042	3043	3048	3058	3059	3069	3074	3076	3078	3088	3098	3098	3118	3118
C	guadagno		-20	-40	-42	-43	-48	-58	-59	-69	-74	-76	-78	-88	-98	-98	-118	-118
C	puntataS		20	20	2	1	5	10	1	10	5	2	2	10	10	20	20	20
M	puntata	20	20	2	1	5	10	1	10	5	2	2	10	10	10	20	20	
C	spinOmaggio														1	0	0	0
C	risultato																	Sara' per un'altra volta!



Type	Functions	State 0	State 1	State 2	State 3	State 4	State 5	State 6	State 7	State 8	State 9	State 10	State 11	State 12	State 13	State 14	State 15
C	rullo1		WILD	RIBES	SPIN	UVA	RIBES	UVA	MELA	JACKPOT	WILD	JACKPOT	RIBES	RIBES	MELA	KIWI	RIBES
C	rullo2		PERA	PERA	RIBES	RIBES	KIWI	KIWI	WILD	JACKPOT	UVA	JACKPOT	MELA	PERA	RIBES	SPIN	UVA
C	rullo3		KIWI	PERA	MELA	SPIN	PERA	RIBES	UVA	PERA	KIWI	JACKPOT	PERA	RIBES	RIBES	SPIN	KIWI

Type	Functions	State 0	State 1	State 2	State 3	State 4	State 5	State 6	State 7	State 8	State 9	State 10	State 11	State 12	State 13	State 14	State 15
C	spin	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C	portafoglio	100	98	88	86	86	86	66	61	56	36	3026	3006	2996	2986	2981	2981
C	cassaSlot	3000	3002	3012	3014	3014	3014	3034	3039	3044	3064	74	94	104	114	119	119
C	guadagno		-2	-12	-14	-14	-14	-34	-39	-44	-64	2926	2906	2896	2886	2881	2881
C	puntataS		2	10	2	20	20	20	5	5	20	10	20	10	10	5	20
M	puntata	2	10	2	2	2	20	5	5	20	10	20	10	10	5	5	5
C	spinOmaggio				1	1	0	0	0	0	0	0	0	0	0	2	1
C	risultato																

Functions	State 15	State 16	State 17	State 18	State 19	State 20	State 21	State 22	State 23	State 24	State 25	State 26	State 27	State 28	State 29	State 30	State 31
rullo1	RIBES	MELA	JACKPOT	MELA	KIWI	SPIN	KIWI	MELA	RIBES	KIWI	PERA	UVA	RIBES	MELA	UVA	UVA	UVA
rullo2	UVA	RIBES	PERA	MELA	PERA	JACKPOT	KIWI	PERA	SPIN	UVA	KIWI	RIBES	RIBES	UVA	WILD	RIBES	RIBES
rullo3	KIWI	KIWI	KIWI	UVA	UVA	PERA	UVA	MELA	SPIN	PERA	RIBES	UVA	KIWI	PERA	KIWI	JACKPOT	JACKPOT

Functions	State 15	State 16	State 17	State 18	State 19	State 20	State 21	State 22	State 23	State 24	State 25	State 26	State 27	State 28	State 29	State 30	State 31
spin	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	30
portafoglio	2981	2981	2976	2975	2974	2969	2969	2964	2959	2959	2959	2958	2957	2955	2953	2948	2948
cassaSlot	119	119	124	125	126	131	131	136	141	141	141	142	143	145	147	152	152
guadagno	2881	2881	2876	2875	2874	2869	2869	2864	2859	2859	2859	2858	2857	2855	2853	2848	2848
puntataS	20	20	5	1	1	5	20	5	5	20	20	1	1	2	2	5	5
puntata	5	5	1	1	5	5	5	5	5	5	1	1	2	2	5	5	5
spinOmaggio	1	0	0	0	0	1	0	0	2	1	0	0	0	0	0	0	0
risultato																	Hai sbancato!