Informatica III - Parte A

RELAZIONE PROGETTI

Conti Lorenzo

Matricola 1046163

Sommario

1. Cyclone

URL Shortener

Per il progetto in cyclone ho deciso di realizzare un piccolo programma che simulasse i tipici "URL Shortener" che possono essere trovati su internet.
L'idea è quella di associare ad un URL un nuovo specifico URL più breve e costruito *ad-hoc* in modo che possa essere condiviso, incorporato in testi o memorizzato evitando le lunghe stringhe di cui solitamente gli indirizzi delle pagine web sono composti.

Un esempio può essere:

`http://cs.unibg.it/gargantini/didattica/info3 → cyc.url/EYGwpt`

Metodo di sviluppo

Per prima cosa, per verificare la fattibilità dell'idea ho realizzato il programma utilizzando il linguaggio C prestando particolare attenzione ad evitare operazioni unsafe ed in seguito ho provato a compilare il risultato ottenuto con tramite cyclone.
Il programma non compilava e presentava numerosi errori. Approcciare al debugging del programma analizzando tali errori è risultato molto complicato in quanto non familiare con la tipologia di messaggi di errore individuati.
Il porting del codice è risultato in questo modo più complicato di quanto immaginassi. Ho deciso quindi di riscrivere completamente il codice adottando da subito la sintassi prevista per un programma completamente safe in cyclone.

Strutture dati

La struttura dati principale è "Url" che ospiterà una coppia URL/URL abbreviato. Entrambi i campi sono stati dichiarati come puntatori fat in modo che possa essere salvata la lunghezza della stringa contenuta e in modo che possa essere applicata aritmetica del puntatore stesso in così da poter scorrere la stringa carattere per carattere.

```
typedef struct {
        char *@fat url;
        char *@fat shorturl;
} Url;
```

Il programma verterà alla manipolazione di due liste:

- una lista contenente gli indirizzi web che si desidera accorciare sottoforma di stringa
- una lista (più ampia) contenente strutture “Url”, ovvero coppie di stringhe

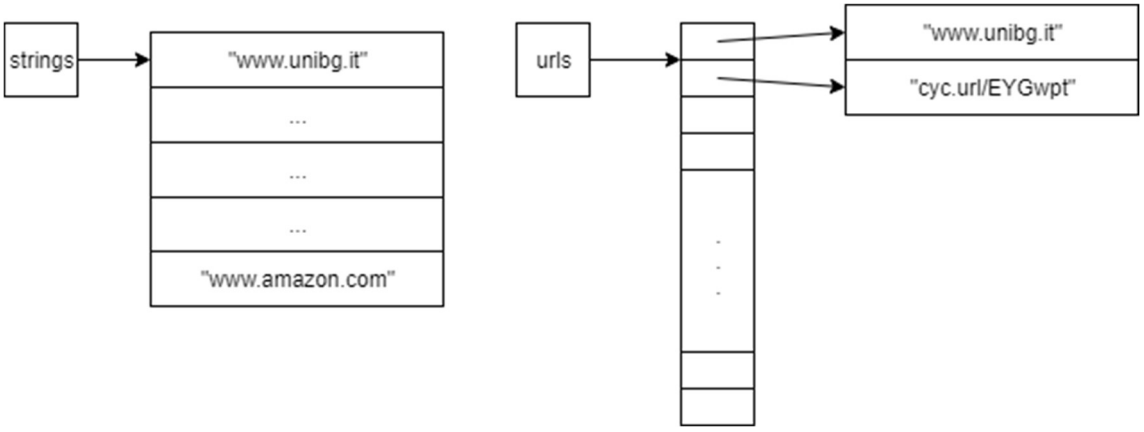


Figura 1 - Strutture Dati: liste

In figura sono mostrate le strutture dati implementate: “strings” e “urls” sono due puntatori ad un ulteriore insieme di puntatori, rispettivamente di stringhe e “Url”.

```
char *@fat*@fat strings = (char *@fat *@fat) calloc(STRING_ARRAY_SIZE, sizeof(char *@fat));
```

```
Url **@notnull @fat urls = (Url ** @fat) calloc(URL_ARRAY_SIZE, sizeof(Url *));
```

Per l’inizializzazione delle liste ho adottato il metodo *calloc* cosi che le liste venissero inizializzate e potessi usare metodi della libreria *string.h* senza ottenere errori per mancata inizializzazione.

Generazione URL

Una volta istanziate le liste, in una prima fase, per ogni stringa contenuta nella lista di stringhe viene creato un nuovo “Url” alla quale viene associato un URL abbreviato creato nel formato *cyc.url/XXXXX*, dove ogni X corrisponde ad un carattere o ad un numero generato in modo casuale nel seguente modo:

Essendo la lista degli “Url” più ampia rispetto a quella delle stringhe, ho reso possibile inserire singoli URL individualmente.

```
initArray(urls, &size, strings);

addUrl(urls, &size, "www.kahoot.com");
```

Il primo metodo permette di inserire nella lista degli “Url” una sequenza di coppie URL/URL abbreviato a partire dalla lista di stringhe passata come terzo argomento.
Il secondo metodo permette di creare una singola istanza di “Url” a partire da una data stringa passata come argomento. Il campo size (passato come riferimento) tiene traccia della quantità di “Url” correttamente salvati nella lista precedentemente allocata ed inizializzata.

Conclusione

Scrivendo il programma adottando la sintassi di cyclone dalla prima riga di codice mi ha permesso di acquisire familiarità di quanto, secondo la mia opinione, avrei acquisito tentando di correggere gli errori dovuti al porting. Le maggiori difficoltà sono state dovute alla gestione dei puntatori doppi utilizzati per mantenere il riferimento alle due liste. Durante la programmazione, cyclone ha mostrato la sua forte rigidezza in termini di sicurezza del codice, a differenza di quanto avvenuto nella stesura del codice iniziale in C. A prova di questo, nonostante i tanti errori e warning incontrati e risolti, rimane un ultimo avvertimento:

```
integral size mismatch; int -> char conversion supplied
```

dovuto alla conversione da intero a char (secondo la tabella ASCII) adottata nella generazione dell’URL abbreviato. Nonostante il cast sia completamente lecito, viene segnalato dal compilatore.
Infine, si può notare che ciascun puntatore creato durante l’esecuzione (con malloc o con calloc) non è mai stato esplicitamente liberato. Questo perché, come descritto nella documentazione, cyclone non permette l’utilizzo di *free* ed adotta un garbage collector.

“Storage in the heap is explicitly allocated via new or malloc, but there is no support in Cyclone for explicitly freeing an object in the heap. The reason is that Cyclone cannot in general track the lifetimes of individual objects within the heap accurately, so it can’t be sure whether dereferencing a pointer into the heap would cause problems. Instead, unless otherwise specified, a conservative garbage collector is used to reclaim the data allocated in the heap.”

2. C++

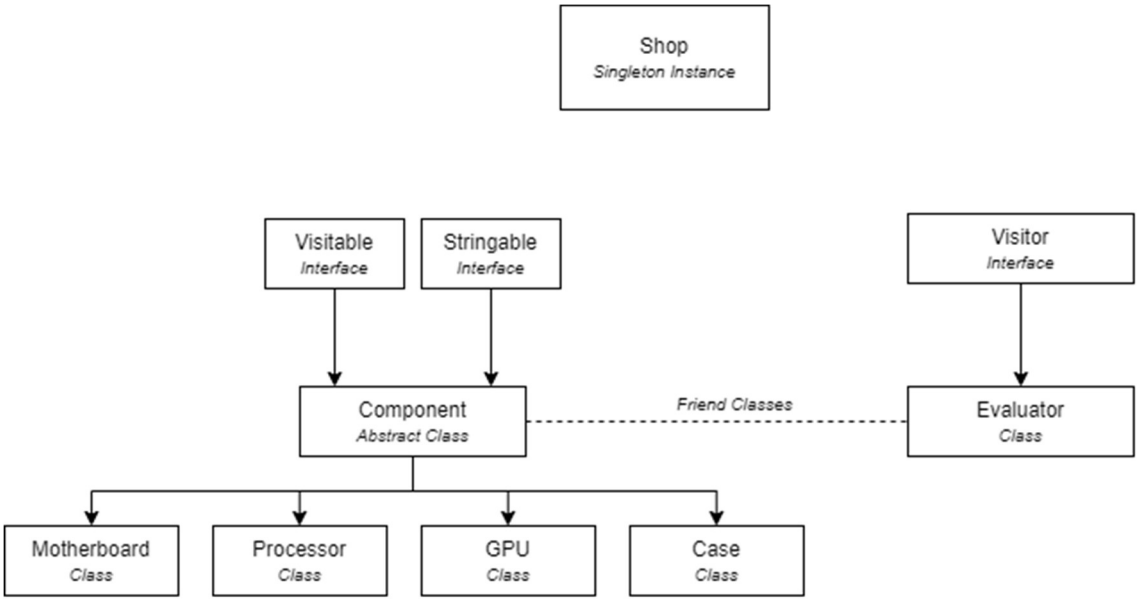
PC Components Shop

Per il progetto in C++ ho realizzato un programma che simulasse il software di un negozio di componentistica per computer, in modo esemplificativo ma che potesse sfruttare quante più caratteristiche possibili del linguaggio.

Nel negozio saranno disponibili differenti tipologie di hardware (processori, schede madri, schede grafiche e case), sarà possibile valutarne le proprietà, valutare il valore del prodotto e verificarne la compatibilità.

Per quanto riguarda il lato implementativo, per ciascuna classe è stato creato un file .h contenente l'intestazione della classe stessa e la definizione dei campi e dei metodi ed un file .cpp contenente l'implementazione. Sono stati adottati i pattern singleton e visitor, sono stati adottati i template e le friend class. In seguito, mostrerò in che modo.

Gerarchia delle classi



Oltre alle classi mostrate in figura, sono presenti ulteriori classi: *Comparator* e *Utilities.* La prima può essere considerata come un "visitor alternativo" mentre la seconda contiene metodi di appoggio allo sviluppo delle funzionalità.

Pattern

Singleton: il negozio è modellato da una istanza singleton *Shop* che contiene due liste: una lista dei prodotti presenti nel negozio ed una lista dei prodotti aggiungi alla propria *build*.
Visitor: essendo la struttura gerarchica delle classi "orizzontale" possiamo trarre enorme vantaggio nell'utilizzare il pattern visitor così che, ogni qualvolta dovesse essere implementato un metodo per ogni componente, possiamo creare un'unica nuova classe contenente tutte le implementazioni del metodo, senza modificare classe per classe.

Nel progetto è stato creato un visitor *Evaluator* che, per ogni componente, calcola il valore del prodotto sulla base delle caratteristiche del prodotto stesso. Ogni componente quindi, avrà una diversa implementazione del metodo.

Per facilitare l'implementazione del visitor, *Component* ed *Evaluator* sono state dichiarate come **friend class.** Ciò permette di avere accesso ai campi privati del componente e di salvare il valore calcolato tramite il valutatore direttamente all'interno della classe.

```
class Visitor {

public:
	virtual void visit(Motherboard *) = 0;
	virtual void visit(Processor *) = 0;
	virtual void visit(Case *) = 0;
	virtual void visit(Gpu *) = 0;

	virtual ~Visitor(){};
};
```

```
class Visitable {

public:
	virtual ~Visitable() { }
	virtual void accept(Visitor * v) = 0;
};
```

Comparator: visitor alternativo

Il pattern visitor adottato, nella sua versione base, pone una limitazione nel momento in cui un metodo ha bisogno di ricevere anche un parametro in ingresso. L'operazione di controllo della compatibilità infatti, oltre a dover essere implementata per ogni componente, deve anche essere implementata nei confronti degli altri componenti.
Un esempio è:

```
bool Comparator::compare(Motherboard * m, Processor *p){
	return m->getSocket().compare(p->getSocket());
 }
```

Come si può notare, la compatibilità del componente *Motherboard* è data dall'uguaglianza di uno dei propri campi con un campo del componente processore passato come argomento. Questo non è possibile in una tipica chiamata di metodo tramite visitor del tipo *accept(object)*.
La classe *Comparator* quindi contiene il metodo compare definito per ogni componente (come nel pattern visitor) con l'unica differenza che non viene invocato con il metodo *visit | accept*.

Utilities Class: template, enum, static e instanceof

```
class Utilities {
public:

	static MoBoShape shapeSelector(std::string const& shape){

		if(shape.compare("ATX")) return ATX;
		if(shape.compare("mATX")) return mATX;
		if(shape.compare("microATX")) return microATX;
		return undefined;
	}

	template <typename T> static std::string stringify(T t){
		std::stringstream s;
		s << t;
		return s.str();
	}

	template <typename Base, typename T> static bool instanceof(T *ptr) {

	    return dynamic_cast<const Base*>(ptr) != nullptr;
	}

	static bool better(Component * c1, Component * c2){

		int v1 = c1->getValue();
		int v2 = c2->getValue();

		if(c1 > c2) return true;
		else if(v1 == v2){
			if(c1->getPrice() < c2->getPrice()) return true;
		}
		return false;
	}
};
```

In figura è mostrata l'implementazione della classe *Utilities* e dei suoi metodi:

- *shapeSelector* permette di ritornare il tipo enumerativo a partire dalla stringa passata come argomento. Verrà utilizzato nel costruttore delle componenti che contengono come proprietà la forma della scheda madre;
- *stringify* è un metodo generico che permette di convertire qualsiasi tipo di variabile in una stringa cosi da poterla concatenare nei parametri utilizzati con *cout*. Verrà usata nei metodi *toString* per concatenare stringhe a numeri;

```
for(i = clist.begin(); i < clist.end(); i++){
		Utilities::instanceof<Motherboard>(*i) ? mobos.push_back(*i) : void();
		Utilities::instanceof<Processor>(*i) ? cpus.push_back(*i) : void();
		Utilities::instanceof<Case>(*i) ? cases.push_back(*i) : void();
		Utilities::instanceof<Gpu>(*i) ? gpus.push_back(*i) : void();
	}
```

- *instanceof* è anch'esso un metodo generico che simula l'operatore instanceof di Java. Attraverso *dynamic_cast* prova ad effettuare cast sul puntatore passato come parametro per ottenere un oggetto della classe *Base*. Nel caso il puntatore ottenuto dal cast fosse nullo, l'oggetto passato

come argomento non è istanza della classe *Base.* Verrà usato nella classe *Shop* per distinguere le istanze di componenti dalla lista di componenti globale;

- *better* è un semplice metodo statico che confronta due componenti sulla base del loro valore.

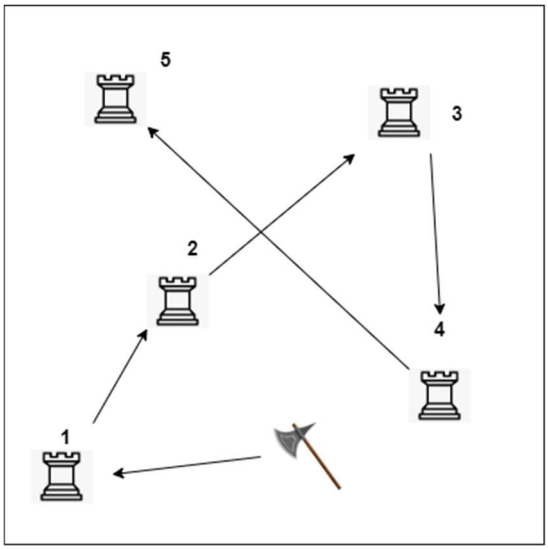
3. Scala

Attackers & Defenders – Scala Minigame

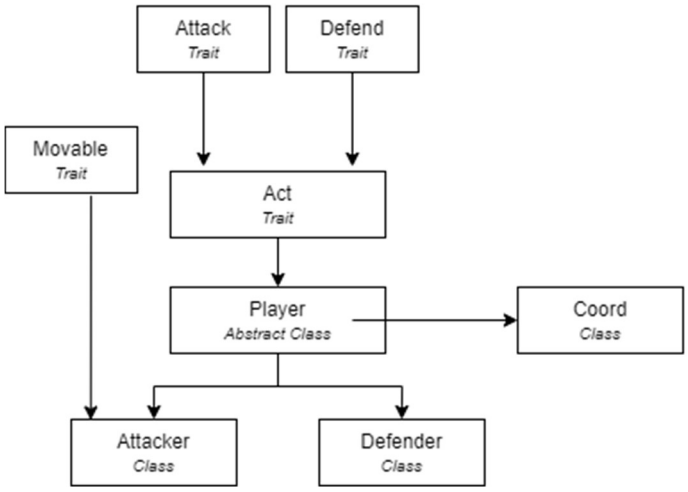
Per il progetto in Scala ho sfruttato i suoi costrutti Object Oriented per realizzare il seguente minigioco:

In un campo di battaglia (una mappa 10x10) vengono schierati cinque giocatori per ognuna delle due squadre: attackers e defenders. Per vincere la partita, gli attackers devono distruggere tutti i defenders schierati che, a loro volta, tenteranno di difendersi.
Ogni giocatore viene schierato in maniera completamente casuale ad una specifica coordinata sulla mappa.

Inizialmente vengono schierati sulla mappa tutti e cinque i defenders e manterranno la loro posizione durante l'intera durata della partita. Verranno poi schierati uno alla volta gli attackers che dovranno in primo luogo cercare di avvicinarsi al primo defender da colpire. Durante il tragitto il defender in questione attaccherà l'attacker, diminuendogli man mano i punti vita.
Una volta che l'attacker ha raggiunto il defender potrà cominciare ad attaccarlo. Distrutto il primo defender si sposterà verso il secondo defender e così proseguendo fino a che tutti i defender sono stati distrutti. Nel caso uno degli attackers morisse in battaglia, verrà schierato un suo compagno, che riprenderà l'opera di distruzione dove era stata interrotta dal compagno (sempre previo raggiungimento del defender da attaccare).



Gerarchia delle classi



Costrutti di scala

Nell'implementazione del progetto ho cercato di utilizzare diversi costrutti messi a disposizione dal linguaggio Scala studiati nel corso in altri linguaggi di programmazione tra cui:

- Definizione di classi e traits

```
trait Movable { def move(distance : Double) : Double}

class Coord(var x : Int, var y : Int){

  def distance(px : Int, py: Int) : Double = sqrt(pow(x-px,2) + pow(y-py,2))
}
```

- *Override* dei metodi

```
override def toString() : String = {
       return name + " HP: " + hp + " DMG: " + dmg + " SPAWNED AT: (" + pos.x + "," +
        pos.y  + ") with RANGE: " + range
  }
```

- Generics: ho implementato un metodo generico richiedendo che uno dei parametri fosse una classe che estendesse il *trait Movable* nel modo seguente:

```
def move[T <: Movable](p : T, dist : Double) : Double = {
  p.move(dist)
}
```

- Passaggio di parametri in un metodo definiti a partire dalle funzionalità che implementano

```
def doAttack(a: {def attack : Int}, d: Player) {
  d.defend(a.attack)
}
```

- *Casting* delle superclassi in sottoclassi per la specializzazione nel passaggio di parametri in una funzione. Infatti, nella definizione della funzione *moveAndAttack* era necessario ricevere come argomento un Attacker ed un Defender, cosi da poter accedere ai loro metodi *attack* e *defend*. Essendo le liste *attackers* e *defenders* definite di tipo generico (e astratto) *Player*, passando *attackers(0)* e *defenders(0)* non avrei potuto accedere a tali parametri.

```
distance = moveAndAttack(attackers(0).asInstanceOf[Attacker],defenders(0).asInstanceOf[Defender],
distance)
```

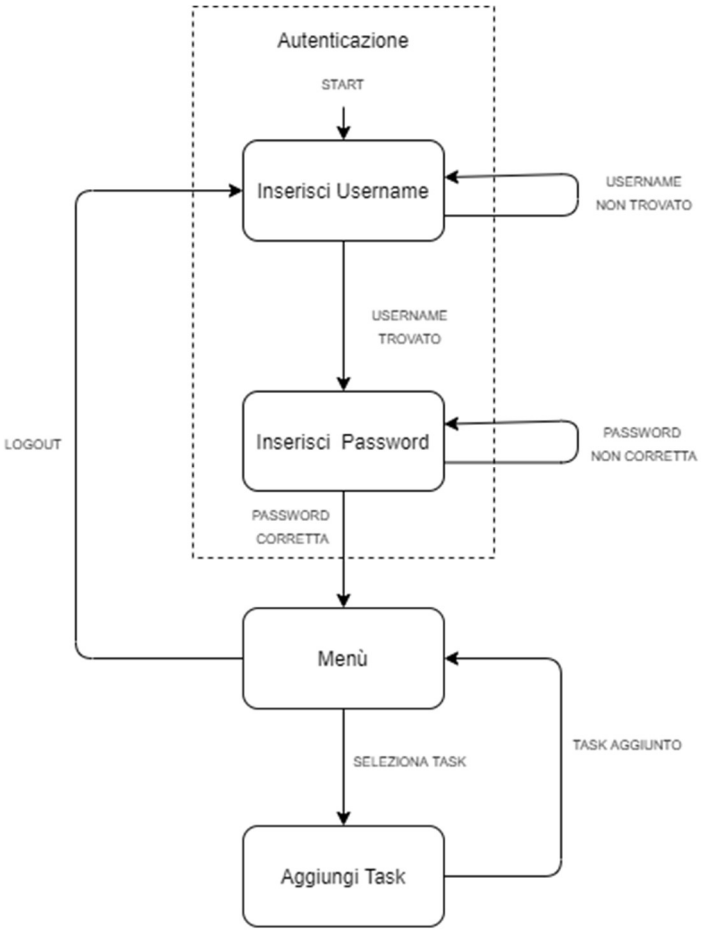
- *Getter* e *Setter*

```
def hp = _hp
def hp_(nhp : Int) {
  println(name + " is taking damage! HP: " + hp + " -> " + nhp)
  _hp = nhp
}
```

4. ASM

TO DO Manager

Per il progetto in ASM ho realizzato una piccola specifica che possa essere vista come un programma “To Do Manager”.
L’idea è che un utente, dopo essersi autenticato, può aggiungere alla propria lista dei task o degli impegni da svolgere durante la giornata.
Una sintesi grafica della macchina a stati può essere schematizzata nella figura seguente:



Il codice è stato scritto utilizzando il tool per Eclpise e in seguito testato con l’animatore. Un esempio è mostrato in seguito.

Domini:

```
enum domain Task = { FARE_SPESA | STUDIARE | RIUNIONE | PARTITA_SCACCHI | ALLENAMENTO }
enum domain Operazione = { INSERISCI_TASK | LOGOUT }
enum domain Stato = { INSERISCI_USERNAME | INSERISCI_PASSWORD | MENU | SELEZIONA_TASK }
abstract domain Utente
```

Il dominio Stato definisce i possibili stati in cui si può trovare l'applicazione, come si può notare dalla figura sopra. Lo stato globale del sistema verrà mantenuto in una variabile (o funzione) dinamica controllata "stato".
In ogni regola che verrà definita, si controllerà quale è il valore dello stato del sistema e si stabilirà se la regola debba essere seguita o meno.
Il dominio Operazione definisce le operazioni che possono essere scelte all'interno del menù e il dominio Task specifica quali azioni possano essere aggiunte alla propria lista di task.
Tramite il dominio astratto Utente invece, sarà possibile definire due utenti ("lorenzo" e "admin") con la quale effettuare il processo di autenticazione e i quali avranno una relativa lista di task.

Funzioni controllate:

```
dynamic controlled print : Any
dynamic controlled stato : Stato
dynamic controlled utenteAttivo : Utente
dynamic controlled listaTask : Utente -> Seq(Task)
```

La variabile *print* verrà usata come contenitore per i messaggi di log mentre *ListaTask* assegnerà a ciascun utente una lista, creata vuota durante la fase di inizializzazione del programma.

Funzioni monitorate:

```
dynamic monitored operazione : Operazione
dynamic monitored username : Utente
dynamic monitored password : String
dynamic monitored task : Task
```

Queste variabili gestiranno l'interazione con l'utente e saranno i contenitori nel quale verranno inserite le informazioni inserite da esso.

Costanti: con la quale definisco i due utenti del sistema

```
static lorenzo : Utente
static admin : Utente
```

Funzione derivata per l'associazione della password al relativo utente:

```
derived getPassword : Utente -> String     // dichiarazione

function getPassword($u in Utente) =       // implementazione
        switch($u)
                case lorenzo : "password"
                case admin : "admin"
        endswitch
```

Pattern seguito per la definizione delle regole:

```
macro rule r_sceltaTask =
        if( stato = SELEZIONA_TASK ) then
                if( exist unique $t in Task with $t = task ) then
                        par
                                listaTask( utenteAttivo ) := append( listaTask( utenteAttivo ), task)
                                print := concat ("task aggiunto alla lista: ", toString( task ) )
                                stato := MENU
                        endpar
                endif
        endif
```

Per prima cosa inserisco la condizione sullo stato in modo che il resto del codice venga eseguito solo se effettivamente ci troviamo nello stato corretto. Dopodichè richiedo all'utente di inserire un valore (nell'esempio di scegliere un task tra quelli disponibili nel dominio) e nel caso il valore inserito fosse corretto eseguo specifiche operazioni.
Tipicamente nelle regole che ho implementato si ha sempre almeno un'operazione per variare lo stato del sistema ed un'operazione di log tramite la variabile monitorata *print*. Nell'esempio mostrato è presente un ulteriore operazione corrispondente all'inserimento del task alla lista personale dell'utente.

Main Rule e inizializzazione:

```
main rule r_Main =
       seq
       r_inserisciUsername[]
       par
              r_inserisciPassword[]
              r_menu[]
              r_sceltaTask[]
       endpar
endseq


// INITIAL STATE
default init s0:
       function stato = INSERISCI_USERNAME
       function print = "Inserire username"
       function listaTask( $u in Utente ) = []
```

State 1	State 2	State 3	State 4	State 5	State 6
lorenzo	lorenzo	lorenzo	lorenzo	lorenzo	
"password"	"password"	"password"	"password"	"password"	
Benvenuto, seleziona l'operazione che vuoi eseguire	Seleziona un task	task aggiunto alla lista: PARTITA_SCACCHI	Seleziona un task	task aggiunto alla lista: RIUNIONE	Logout effettuato con successo
MENU	SELEZIONA_TASK	MENU	SELEZIONA_TASK	MENU	INSERISCI_USERNAME
lorenzo	lorenzo	lorenzo	lorenzo	lorenzo	lorenzo
INSERISCI_TASK	INSERISCI_TASK	INSERISCI_TASK	INSERISCI_TASK	LOGOUT	
	PARTITA_SCACCHI	PARTITA_SCACCHI	RIUNIONE	RIUNIONE	
		[PARTITA_SCACCHI]	[PARTITA_SCACCHI]	[PARTITA_SCACCHI,RIUNIONE]	[PARTITA_SCACCHI,RIUNIONE]