Progetti di Informatica III-A

Enrico Bacis

- Università degli Studi di Bergamo -

Versione del Documento: **1.0** - 20130219

Indice

Parte I
Cyclone

Cyclone è un dialetto safe del C che permette di prevenire diversi tipi di errori e problemi di sicurezza molto comuni in C come buffer overflow, stringhe non terminate e dangling pointers.

Per ottenere questi risultati il linguaggio C è stato esteso con caratteristiche come il *garbage collector*, che solleva il programmatore dal dover esplicitamente deallocare la memoria con le chiamate **free()**, riducendo la possibilità di incorrere in dangling pointer o di memoria non deallocata al termine del suo utilizzo; il garbage collector infatti libera automaticamente la memoria utilizzata da oggetti di cui non esistono più puntatori (e che quindi sono safe da cancellare).

Altra caratteristica importante di Cyclone sono i qualificatori dei puntatori che meglio specificano i possibili valori assunti dai puntatori e aggiungono controlli sull'utilizzo degli stessi. In questo modo Cyclone permette di eseguire in sicurezza operazioni che riguardano i puntatori come aritmetica sui puntatori e gestione di stringhe. Nel seguito della relazione verranno presi in esame i diversi qualificatori e si illustrerà come sono stati usati nel progetto.

1 Descrizione del progetto

Il progetto sviluppato è una libreria per la crittografia di stringhe utilizzando cifrari a sostituzione con chiavi decise dall'utente.

La libreria non è stata sviluppata con un solo cifrario in mente, ma è pensata in maniera modulare, in modo che sia facile implementare l'utilizzo di diversi cifrari di sostituzione e anche crearne di nuovi.

Come esempi sono state scritte le funzioni per cifrare testi utilizzando il cifrario di Cesare e il cifrario ROT13 oltre al caso di chiave decisa dall'utente.

Oltre a questo sono anche state re-implementate in Cyclone alcune funzioni di supporto per le stringhe come strcpy e strncpy in maniera safe a differenza delle loro controparti in C.

1.1 Scelta e motivazioni

E' stato scelto di creare una libreria e non un programma perché trattandosi di un piccolo progetto, di seguire l'approccio migliore per la stesura di un programma specifico, ovvero quella di creare un insieme di funzionalità che non siano dipendenti dal programma stesso ma che possano essere riutilizzate anche in altri progetti.

E' stata dedicata particolare attenzione anche alla suddivisione delle funzioni in modo da seguire un approccio DRY[1] e una buona architettura del software individuando i casi generali e utilizzando questi nella codifica delle funzioni a più specifiche.

Il codice è stato scritto direttamente in Cyclone senza effettuare porting, per vedere quanto sforzo sia necessario per un programmatore C iniziare un nuovo progetto avendo cura di scriverlo in un linguaggio Safe come Cyclone. Si è quindi voluto verificare quanto la stesura di codice nativo in Cyclone sia più complessa di quella in C (non è infatti possibile pensare di scrivere nuovi applicativi avendo una fase di scrittura in C e una di correzione in Cyclone).

Vista la semplicità di utilizzo delle caratteristiche di Cyclone e la sua somiglianza con C, lo sforzo richiesto al programmatore è minimo e i concetti vengono assimilati in breve tempo.

[1] Don't Repeat Yourself

1.2 Funzionalità principali

1.2.1 Utilities

min, *abs*, *islower*, *isupper*, *isletter*, *isdigit*

1.2.2 String Utilities

- ***safe_ strncpy***

  Descrizione: Copia n caratteri dalla stringa sorgente alla destinazione, o meno se la destinazione è più piccola.

  Parametri: *char * @notnull @fat dst, const char * @notnull @fat src, unsigned int n*

  Tipo di ritorno: *int*

- ***safe_ strcpy***

  Descrizione: Copia tutto il contenuto della stringa sorgente in quella destinazione, o meno se la destinazione è più piccola.

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

  Tipo di ritorno: *int*

- ***strclone***

  Descrizione: Alloca una nuova stringa della dimensione della stringa sorgente e ve ne copia il contenuto.

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

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

- **strswap**

  Descrizione: Restituisce una nuova stringa che è il risultato dell'esecuzione della lista di swap applicati alla stringa sorgente.

  Parametri: *const char * @notnull @fat src, int * @notnull @numelts(2) ?swap*

  – Il secondo parametro è un array di array di 2 elementi, verrà meglio descritto successivamente nella sezione relativa all'utilizzo dei **Bounded Pointers**.

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

1.2.3 Cipher Utilities

- ***charencrypt***

  Descrizione: Ritorna il carattere *c* (se alfabetico) nella sua controparte criptata secondo l'algoritmo di sostituzione con offset *off* in avanti.

  Parametri: *char c, int off*

  Tipo di ritorno: *char*

- **strencrypt**

  Descrizione: Ritorna una nuova stringa nella quale i caratteri alfabetici sono criptati secondo l'algoritmo di sostituzione con chiave *key*. Se la chiave è più corta della stringa, essa viene re-iterata, se la chiave è vuota, viene restituito un errore a schermo (non

bloccante) e ritornata una stringa vuota. Questo tipo di codifica è nota sia come codifica a sostituzione che col nome di Codifica di Vigènere.

Parametri: *const char * @notnull @fat str, int * @notnull @fat key*

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

- **strdecrypt**

  Descrizione: Ritorna la stringa decrittata con chiave key (vedi strencrypt).

  Parametri: *const char * @notnull @fat str, int * @notnull @fat key*

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

- **str2key**

  Descrizione: Restituisce un array di interi a partire da una stringa. La conversione è una ASCII shiftata in modo da preservare il fatto che il carattere '0' sia mappato sul numero 0. Questa funzione è utile per genere chiavi per la funzione strencrypt a partire da stringhe.

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

  Tipo di ritorno: *int ?*

1.2.4 Ciphers

- **cesar**

  Descrizione: Restituisce la stringa cifrata con il cifrario di Cesare con offset *off*.

  Parametri: *const char * @notnull @fat str, int off*

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

- **rot13**

  Descrizione: Restituisce la stringa cifrata con il cifrario ROT13[2]

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

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

- **derot13**

  Descrizione: Restituisce la stringa decifrata con il cifrario ROT13

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

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

[2]Caso specifico del cifrario di Cesare con offset 13. it.wikipedia.org/wiki/ROT13

2 Costrutti Cyclone utilizzati

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

2.1 Puntatori *

Cyclone permette l'utilizzo di normali puntatori * con le seguenti modifiche rispetto a 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)
  - In realtà quando viene richiesta si procede generando un warning, effettuando un cast ad un puntatore @fat con size 1 (vedi qualificatore @fat) e quindi successivamente avremo un errore in runtime.

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

2.2 Qualificatore @notnull (puntatore @)

Controllare ad ogni de-reference che il puntatore non sia nullo può essere dispendioso. Con questo qualificatore possiamo effettuare il controllo all'assegnamento del valore e evitarlo al suo utilizzo. E' uno dei più utili ed utilizzati qualificatori di Cyclone insieme a @fat.

E' stato utilizzato praticamente ovunque nel codice come si può vedere nella sezione Funzionalità.

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

2.3 Qualificatore @fat (puntatore ?)

Questo qualificatore impone che il puntatore su cui viene applicato mantenga anche l'informazione sul numero degli elementi dell'array. Questo dato è accessibile utilizzando la funzione **numelts(ptr)**.

- Permettono aritmetica sui puntatori (con controllo che non si esca dall'array)
- Tutti gli array possono essere convertiti a @fat (e generalmente viene fatto essendo molto utile).
- Conversione da * a ? automatica con size=1 e warning
- Conversione da ? a * non problematica (bounds check)
- Conversione da ? a @ con bounds check e null check

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

Come per il puntatore @notnull, anche questo è stato utilizzato praticamente ovunque nel codice, ogni qualvolta ci fosse una stringa.

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

Esempio con array di interi:

```
int ?str2key(const char * @notnull @fat str) {
    return new { for i < numelts(str)-1: *(str+i) - '0' };
}
```

In questo caso non è stato utilizzato anche il qualificatore @notnull perché è possibile che la chiave sia nulla. Il problema infatti non sta nel creare una chiave nulla, che deve essere possibile, ma nel suo utilizzo.

2.4 Qualificatore @zeroterm

Qualificatori utilizzati per indicare che gli array a cui puntano sono terminati da caratteri \0. 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*.

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

2.5 Bounded Pointers - Qualificatore @numelts(n)

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.

E' abbastanza difficile trovare come utilizzarlo, ma con un buon utilizzo si presta anche a sostituire delle struct create ad hoc per certe situazioni.

Nel progetto ad esempio è stato usato per indicare che l'argomento della funzione strswap è un'array di array di due elementi, quindi un array di coppie (gli elementi della stringa su cui verrà effettuato lo swap). Un comportamento simile in C era ottenibile solamente creando una struttura "coppia" che contenesse due interi.

```
char * @notnull @fat strswap(const char * @notnull @fat src, int * @notnull @numelts(2) ?swap) {
    char * @notnull @fat dst = strclone(src);
    for (int i = 0; i < numelts(swap); i++) {
        dst[swap[i][0]] = src[swap[i][1]];
        dst[swap[i][1]] = src[swap[i][0]];
    }
    return dst;
}
```

Con l'utilizzo di questi puntatori si possono anche creare funzioni che ricevono come parametri un array di dimensioni non note a compile time e la sua dimensione nel modo:

```
int f(tag_t num, int [num])
```

Questo tuttavia non sembra essere molto utile visto che si possono utilizzare i puntatori @fat.

2.6 calloc() e Garbage Collector

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

```
char * @notnull @fat strclone(const char * @notnull @fat src) {
    int n = numelts(src);
    char * @notnull @fat dst = calloc(n, sizeof(char));
```

```
    int m = safe_strcpy(dst, src);
    return dst;
}
```

Si è fatto utilizzo del garbage collector, infatti tutte le variabili allocate nello heap non sono liberate attraverso delle chiamate a **free()** ma sono lasciate gestire in maniera automatica dal Garbage Collector, che si occuperà di liberare la memoria quando le variabili non sono più referenziate da alcun puntatore.

2.7 Array Comprehension

E' stato piacevole trovare in Cyclone la funzionalità di Array Comprehension, che permette di scrivere, o meglio descrivere delle liste a partire dalle loro relazioni con i numeri da 0 a n. Ad esempio si può ottenere la lista dei numeri pari da 0 a 200 moltiplicando per 2 i numeri da 0 a 100. Questa caratteristica è stata spesso usata nel progetto al posto dei cicli for, in modo che il codice sia più facile ed intuitivo.

```
char * @notnull @fat strdecrypt(const char * @notnull @fat str, int * @notnull @fat key) {
        return strencrypt(str, new { for i < numelts(key): -key[i] });
}
```

Questa funzione è simile alle List Comprehension in Python, anche se queste ultime sono di gran lunga più potenti.

2.8 let

Il costrutto **let** permette di dichiarare una variabile e assegnarvi un valore senza dover specificare il tipo della variabile; infatti verrà fatta inferenza sul tipo del valore assegnato per poi dichiarare la variabile in modo che possa contenere quel dato. Questo comportamento è da un certo punto di vista simile a quello adottato dai linguaggi con dynamic typing e risulta molto utile in un linguaggio come Cyclone dove scrivere il tipo della variabile con tutti i suoi qualificatori può essere dispendioso. In questo modo si lascia al compilatore la scelta del tipo adatto sollevando il programmatore dalla scelta e velocizzando l'utilizzo. Si può scrivere quindi:

```
let str = rot13("hello world");
```

invece che:

```
char * @notnull @fat str = rot13("hello world");
```

3 Supporto allo sviluppo

Il supporto allo sviluppo è stato una parte centrale del lavoro ed ha impegnato diverso tempo per poter trovare dei modi che rendano l'utilizzo di Cyclone più facile anche per altri studenti del corso. Cyclone infatti è stato purtroppo discontinuato e il suo utilizzo è reso difficile dai seguenti problemi:

1. Necessita la compilazione dei sorgenti per funzionare con GCC 4 e superiori
2. Non funziona su architetture a 64 bit
3. Funziona solo su piattaforme UNIX (e Windows utilizzando emulatori di shell UNIX come CYGWIN)

Ognuno di queste voci ha dato spunto a delle riflessioni che hanno portato a dei sottoprogetti (a volte più corposi dei progetti stessi) che si spera possano servire per i futuri studenti del corso.

3.1 Cyclone su Ubuntu

Per ovviare al problema di compilazione con GCC 4 possibile solo con i sorgenti scaricabili dal repository SVN di Cyclone, questi ultimi sono stati compilati per Ubuntu 32 bit e pacchettizzati in un pacchetto software .deb che necessita solo di un doppio click per installare l'intera toolchain Cyclone. Il pacchetto è già stato fornito al docente e utilizzato da diversi studenti (è infatti attualmente il metodo più facile per utilizzare Cyclone).

3.2 Autocyc [BASH]

Basandosi sul sistema inotify che permette di eseguire delle operazioni automaticamente quando si modifica un file, la sincronizzazione con Ubuntu One (software simile a Dropbox con sincronizzazione immediata via LAN) e il linguaggio di scripting BASH per la shell di Linux, è stato scritto un piccolo tool, **autocyc**, che permette di controllare un file, compilarlo e, in caso la compilazione sia positiva, eseguirlo automaticamente.

Questo procedimento è utile quando si vuole programmare in Cyclone su una macchina a 64 bit, compilando automaticamente su un'altra macchina, fisica o virtuale, a 32 bit ogni volta che il file viene modificato.

```
#!/bin/bash

src=${1:-hello.cyc}
out=autocycout
i=1

echo -e "Now watching $src for changes..."

while inotifywait -q -e modify -e attrib $src
do
        clear
        echo -e "[Revision $i]"
        cyclone -o /tmp/$out $src && chmod +x /tmp/$out && /tmp/$out
        rm -rf /tmp/$out
        i=$(($i + 1))
done
```

4 Cyclone Remote Compiler [Bonus]

Nonostante i due sforzi descritti precedentemente per permettere a me e ad altri di compilare correttamente codice Cyclone sulle proprie macchine, non ero ancora soddisfatto della semplicità di utilizzo, e non prospettando previsioni rosee per il porting a 64 bit di questo linguaggio ho deciso di creare **Cyclone Remote Compiler**.

Cyclone Remote Compiler è una applicazione web che permette a chiunque voglia scrivere del codice in Cyclone di compilarlo remotamente, senza dover installare nulla sul proprio PC se non un Browser Web e in maniera indipendente dall'architettura e dal sistema operativo utilizzato.

4.1 Funzionalità

Cyclone Remote Compiler è, al momento della stesura di questo documento, giunto alla versione 1.5 e permette di:

- Compilare un singolo file .cyc di dimensioni fino a 200 kB.
- Tempo massimo di compilazione e di esecuzione 5+5 secondi, dopo i quali l'applicazione si presume in loop e viene terminata segnalando il motivo dell'arresto.
- Visualizzazione dell'output di compilazione e, se non vi sono errori, di quello di esecuzione.
- Mantenimento della path del file da caricare durante la sessione, in questo modo per ricompilare basta modificare il file e cliccare su "compile" senza dover riselezionare il file.
- Last but not the least: cura grafica dell'applicazione con animazioni e syntax highlight del codice sorgente inviato.

4.2 Tecnologie

Le tecnologie utilizzate per questo progetto sono state:

HTML 5, CSS 3.0, Javascript e jQuery Per la parte di presentazione e animazione della pagina web e per il controllo del file sorgente.

AJAX e XHR2 Per l'invio del file sorgente in maniera trasparente all'utente.

Apache e PHP Per l'elaborazione del file sorgente, per richiamare il compilatore, eseguire il compilato e restituire i risultati di compilazione ed esecuzione.

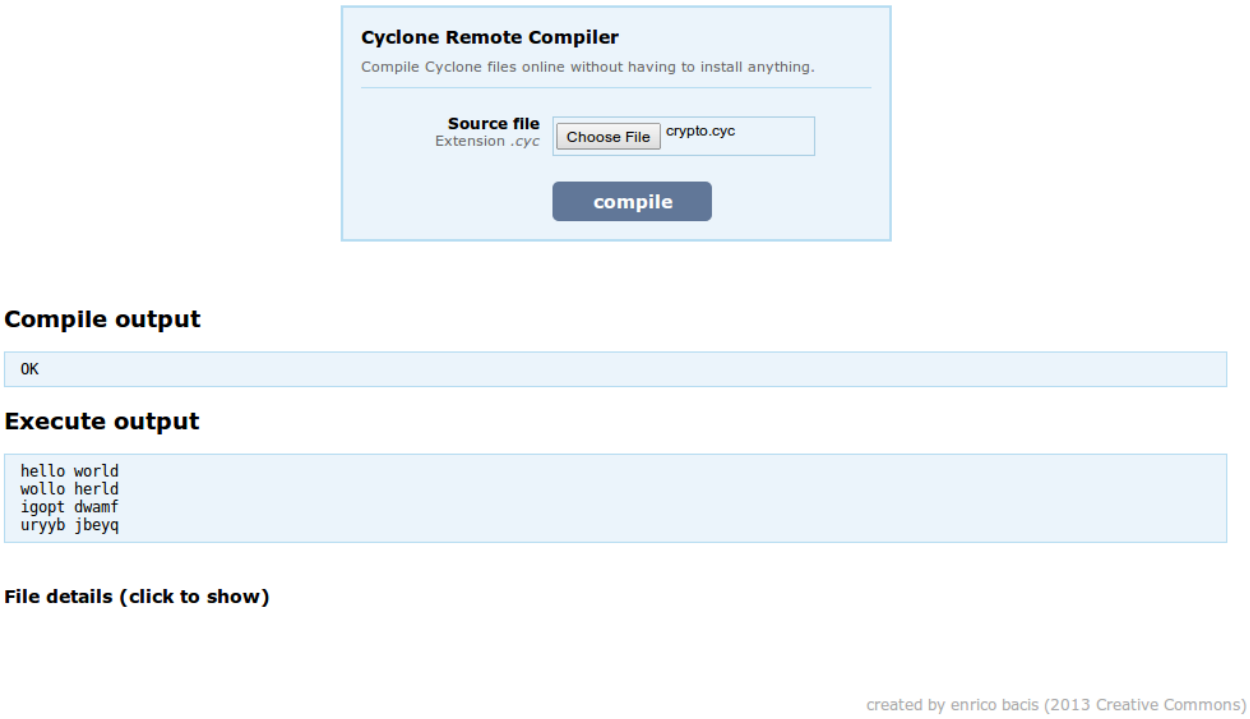
Github Per l'hosting del codice sorgente di Cyclone Remote Compiler.

AWS Amazon Web Services per l'hosting temporaneo dell'applicazione.

Google Prettify Per la funzionalità di Syntax Highlighting del codice sorgente visualizzato.

NO-IP Per l'assegnazione di un indirizzo pubblico per raggiungere la macchina virtuale hostata su Amazon Web Services.

4.3 Conclusione



Cyclone Remote Compiler è disponibile su github per chiunque voglia contribuire e, al momento della stesura di questo elaborato, è raggiungibile al sito:

http://cyclone.hopto.org

Si spera che Cyclone Remote Compiler possa diventare un valido strumento di aiuto all'insegnamento, allo sviluppo e al testing di Cyclone.

Parte II
C++

1 Descrizione del progetto

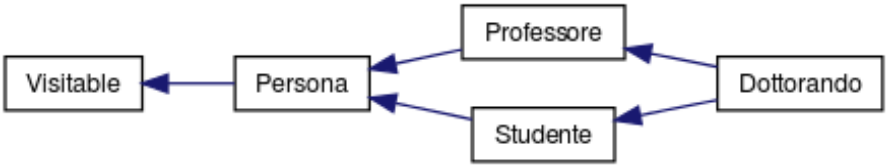
Il progetto sviluppato rappresenta la struttura di una Università, con tre tipi di persone che vi interagiscono:

Studente può conseguire delle votazioni relative ai corsi.

Professore può insegnare dei corsi.

Dottorando riunisce in sé sia le caratteristiche dello studente che quelle del professore.

Tutte e tre estendono (nel caso di Dottorando in maniera indiretta) la classe astratta Persona.



Quasi tutte le funzionalità sono svolte attraverso la classe University, che permette di immatricolare nuovi studenti, docenti e dottorandi.

1.1 Obiettivi

Durante lo sviluppo di questo progetto si ha avuto come obiettivi, più che l'effettiva utilità del progetto svolto, l'aspetto didattico che questo aveva nell'apprendimento di gran parte delle particolarità di C++. Quindi i punti chiave sono stati:

- Utilizzo di gran parte delle caratteristiche (anche quelle considerate più particolari) del C++ cercando di utilizzarle dove risolvano degli effettivi problemi, e non cercando di utilizzarle per il solo gusto di usarle (analizzeremo poi in dettaglio come e dove sono state utilizzate).
- Utilizzo di una vasta gamma di **Design Patterns** (anche questi verranno presi in esame successivamente).

2 Classi Principali

University (*university.h*, *university.cpp*) – Rappresenta l'università e contiene metodi utili alla sua gestione.

Persona (*persona.h*, *persona.cpp*) – Rappresenta una persona, è una classe astratta, quindi non-instanziabile, ma contiene i membri caratteristici di una persona per fornire una base comune alle sue sottoclassi.

Studente (*studente.h*, *studente.cpp*) – Sottoclasse di Persona, rappresenta uno studente universitario, con una mappa di voti.

Professore (*professore.h*, *professore.cpp*) – Sottoclasse di Persona, rappresenta un Professore, con una lista di corsi insegnati.

Dottorando (*dottorando.h*, *dottorando.cpp*) – Sottoclasse sia di Studente che di Professore, rappresenta un Dottorando, che fonde le caratteristiche di Professore e di Studente.

VotiContainer (*voticontainer.h*, *voticontainer.cpp*) – E' una mappa che permette di salvare i nomi dei corsi per i quali lo studente ha conseguito una valutazione almeno sufficiente.

Voto (*voto.h*, *voto.cpp*) – Coppia di valori che rappresentano il nome del corso e la valutazione conseguita dallo studente.

Visitor (*visitor.h*) – Classe astratta di base per tutti i visitor pattern che si vogliono costruire sulla gerarchia di Persona. Contiene metodi virtuali puri **visit** in overload per permettere il double dispatch.

Visitable (*visitable.h*) – Classe astratta di base per tutte le classi che vogliono poter essere visitate dai visitor. Contiene il metodo virtuale **accept** che permette alla classe di accettare i visitor.

Discounter (*discounter.h*, *discounter.cpp*) – Esempio di visitor parametrizzato (si veda la trattazione nella sottosezione visitor pattern) che visita la gerarchia persona e ottiene la percentuale di sconto che deve essere applicata per l'utilizzo dei servizi universitari (come ad esempio la mensa) da parte della persona che accetta il visitor.

2.1 Altri File

main (*main.cpp*) È praticamente una demo sull'utilizzo delle classi. Contiene il metodo main che viene eseguito all'esecuzione del file compilato.

utilities (*utilities.h, utilities.cpp*) Piccola libreria di metodi utili a differenti classi, come un metodo per verificare se in una stringa è presente una sottostringa in maniera case insensitive.

3 Principali Costrutti Utilizzati

3.1 Classi

Ovviamente tutto il progetto fa largo uso della programmazione ad oggetti a utilizza costrutti quali:

- Costruttori, costruttori multipli e distruttori che puliscono gli oggetti creati nello heap
- Membri e metodi pubblici, protetti e privati

Inoltre come particolarità del C++ si è anche utilizzato il costrutto **friend** che permette ad una classe o ad un metodo di una classe di accedere anche ai membri e metodi privati e protetti di un'altra classe.

3.2 Ereditarietà

In C++ quando si ha eredità pubblica, allora si ha anche sottotipazione. Nel codice si trovano molto spesso ereditarietà pubbliche, questo è l'esempio del legame tra Persona e Studente.

3.2.1 Ereditarietà Privata

In alcuni casi potremmo però voler estendere una classe senza rendere pubblici i membri di quest'ultima. In questo modo, analogamente a quanto poi si farà con il pattern Getter/Setter (spiegato dopo) si può ottenere:

- Controllo degli accessi ai membri e metodi della classe ereditata
- Wrapping dei metodi per effettuare controlli di validità sui dati

Nel progetto l'ereditarietà privata è usata in due classi. Vediamo un estratto dalla classe *Voto*, qui si è voluto limitare l'accesso ai metodi della classe std::pair, evitando la modifica del dato dopo la sua scrittura:

```
class Voto: private std::pair <std::string, unsigned short> {
public:
        Voto (std::string const& corso, unsigned short voto);
        virtual ~Voto ();
        std::string const& getCorso () const;
        unsigned short getVoto () const;
};
```

Nella classe *VotiContainer* si è anche voluto ri-pubblicizzare alcuni metodi (questo viene fatto con la keywork **using**). Vediamone un estratto:

```
class VotiContainer: private std::map <std::string, unsigned short> {
public:
        size_t registraVoto (Voto* voto);
        unsigned short getVoto (std::string const& corso) const;
        // Ripubblicizzo i metodi che vondo rendere pubblici
        using std::map<std::string, unsigned short>::const_iterator;
        using std::map<std::string, unsigned short>::size;
};
```

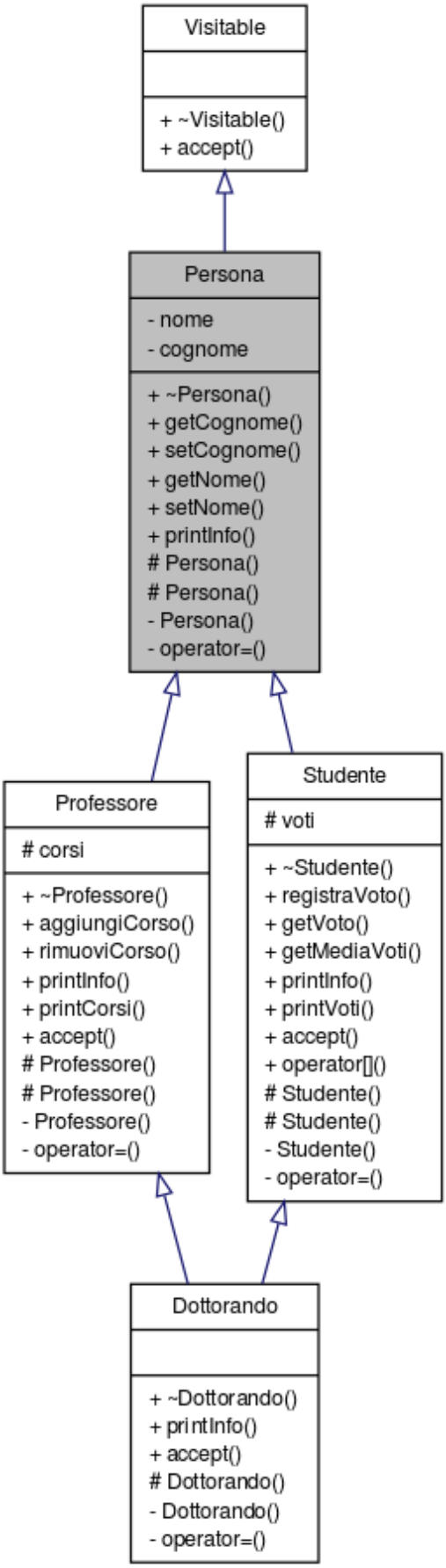
3.2.2 Ereditarietà Multipla e Diamond Problem

Una delle caratteristiche fondamentali di C++ è la possibilità di ereditare da più classi comportamenti. In questo modo il grafo dell'ereditarietà delle classi non è più un albero semplice ma una rete.

Tuttavia l'ereditarietà multipla può portare ad alcuni problemi, infatti se si eredita da due classi che hanno tra le loro basi una stessa classe, ci si troverebbe con due istanze della stessa base class. Il diamond nella nostra gerarchia si chiude con l'inserimento della classe Dottorando:

```
class Dottorando: public virtual Studente, public virtual Professore { ... };
```

Il diamante si vede chiaramente dalla rappresentazione UML della gerarchia:



Il problema del diamond si risolve con l'ereditarietà virtuale, basta infatti che Studente e da Professore ereditino virtualmente Persona, in questo modo garantiamo che ci sia al più una istanza di Persona anche in caso di diamonds:

```
class Studente: public virtual Persona { ... };
class Professore: public virtual Persona { ... };
```

3.2.3 Overriding e Metodi Virtuali

In C++, anche in presenza di ereditarietà e di condizioni valide per fare l'override di un metodo, di default non viene effettuato override ma overload del metodo, e quindi il risultato non è polimorfico: in compilazione non viene scelta la segnatura ma il metodo da chiamare, in base quindi al tipo del riferimento e non al tipo dell'oggetto. Quando invece si vuole fare override è necessario che la base class dichiari il metodo come **virtual**.

Un esempio nel progetto è il metodo *virtual void printInfo()*, implementato in tutte le classi della gerarchia. In questo modo il metodo chiamato in runtime dipenderà dal tipo dell'oggetto e non dal tipo del riferimento. Quindi scrivendo:

```
Persona* p = new Studente();
p->printInfo();
```

verrà chiamato il metodo printInfo di Studente (se il metodo non fosse stato virtual sarebbe stato chiamato quello di Persona).

Se da un metodo di Studente vogliamo richiamare un metodo di Persona su cui è stato fatto override possiamo scrivere:

```
Persona::printInfo();
```

3.2.4 Classi Astratte e Metodi Virtuali Puri

Per poter dichiarare classi astratte (non instanziabili) in C++ dobbiamo fare ricorso a metodi **pure virtual** che sono codificati come:

```
virtual returnType method (args) = 0;
```

Le classi astratte nel progetto sono:

- Visitor

  ```
  virtual void visit (Studente*) = 0;
  virtual void visit (Professore*) = 0;
  virtual void visit (Dottorando*) = 0;
  ```

- Visitable

  ```
  virtual void accept (Visitor* visitor) = 0;
  ```

- Persona

  ```
  virtual ~Persona() = 0;
  ```

  Persona ha anche un secondo metodo virtuale pure che è quello ereditato da Visitable che però non estende.

Le classi Visitor e Visitable non hanno implementazioni, quindi sono come delle interfacce Java.

3.3 Overload

Anche l'overload è un concetto fondamentale, un esempio è quello appena visto per la classe Visitor che ha tre metodi visit con tre metodi visit che però hanno differenti argomenti. Si noti che in questo modo si presenta tuttavia il problema del **Double Dispatching**, che viene risolto poi nella sezione che spiega in dettaglio il Visitor Pattern.

3.3.1 Overload degli operatori

Altra caratteristica del C++ è quella di poter fare overload di quasi tutti i più comuni operatori.

Overload di [] Utilizzato ad esempio della classe Studente come alias della chiamata a *getVoto (string const& corso)*:

```
unsigned short Studente::operator[] (string const& corso) {
        return getVoto(corso);
}
```

Overload di = Utilizzato per definire il comportamento dell'assegnamento. Nel progetto si usa per inibire l'assegnamento alle istanze di alcune classi come University (il metodo viene dichiarato ma mai implementato, in modo che il compilatore generi un errore se viene utilizzato).

Overload di << Questo operatore è molto utile insieme agli output stream come **cout**. In questo modo si può definire il comportamento a fronte di una chiamata *cout << istanza_della_classe*. Vediamo l'esempio di VotiContainer:

```
class VotiContainer: private std::map <std::string, unsigned short> {
        [ ... ]
        friend std::ostream& operator<< (std::ostream& os, VotiContainer const* v);
};

ostream& operator<< (ostream& os, VotiContainer const* v) {
        for (map<string, unsigned short>::const_iterator i = v->begin(); i != v->end(); ++i)
                os << i->second << "\t" << i->first << endl;
        return os;
}
```

Si noti che si è voluto utilizzare un metodo esterno alla classe ma definito come friend della stessa, in modo da poter vedere tutti i suoi membri. Viene ritornato lo stream stesso per poter concatenare elementi come:

```
cout << "value = " << istanza_della_classe << "!"
```

3.4 STL

Nel progetto si è fatto un largo utilizzo di tutti gli aspetti delle STL per ottenere funzionalità avanzate senza dover codificarle e senza quindi rischiare di immettere errori o cali di prestazioni non necessari.

3.4.1 Strutture

Le strutture dati che sono state utilizzate sono:

vector Utilizzato nella classe Professore per mantenere una lista di *std::string* (i corsi insegnati).

unordered_map Utilizzato da University per mantenere una mappa delle persone. La chiave della mappa è la matricola, mentre il valore è il puntatore alla Persona.

map Esteso da VotiContainer per tener traccia (in maniera ordinata secondo registrazione) dei voti dello studente. La chiave è il nome del corso, mentre il valore è il voto.

pair Esteso da Voto che rappresenta poi la coppia chiave/valore inserita in VotiContainer.

3.4.2 Iteratori

Utilizzo degli iteratori per creare algoritmi come quello per il calcolo della media in VotiContainer:

```
float VotiContainer::getMedia () {
        if (this->empty())
                return 0.0f;
        int sum = 0;
        for (map<string, unsigned short>::const_iterator i = this->begin(); i != this->end(); ++i)
                sum += i->second;
        return ((float) sum) / size();
}
```

Altri esempi di utilizzo di iteratori si ritrovano in tutto il codice per scorrere ma anche cercare i dati nelle strutture STL.

3.4.3 Algoritmi

Un esempio di algoritmo STL è il metodo **transform** utilizzato per trasformare in lowercase tutti i caratteri di una stringa prima di poter fare la ricerca di una sottostringa in maniera case-insensitive. Esempio da utilities:

```
bool strfindi(std::string src, std::string str)
{
        std::transform(src.begin(), src.end(), src.begin(), ::toupper);
        std::transform(str.begin(), str.end(), str.begin(), ::toupper);
        return (src.find(str) == std::string::npos) ? false : true;
}
```

4 Pattern

Un aspetto molto importante, se non fondamentale, dello sviluppo del progetto è stata l'attenzione alla comprensione e all'utilizzo dei design pattern architetturale.

4.1 Getter/Setter Pattern

Tutti i membri "interessanti" delle varie classi sono definiti comunque privati e sono acceduti da metodi getter e setter in modo da poter definire i tipi di azioni permesse e fare controllo dei valori impostati. Vediamo un esempio di limitazione degli accessi classe Voto (lettura permessa ma senza scrittura):

```
class Voto: private std::pair <std::string, unsigned short> {
public:
        std::string const& getCorso () const;
        unsigned short getVoto () const;
};
```

Un altro esempio dalla classe VotiContainer sul controllo dei valori:

```
size_t VotiContainer::registraVoto (Voto* voto) {
        if (voto->getVoto() < 18 || voto->getVoto() > 30)
                cout << "È possibile registrare solo voti 18 <= x <= 30";
        else
                this->insert(*voto);
        return this->size();
}
```

4.2 Singleton Pattern

La classe *University* ha un costruttore privato, un membro statico privato, che rappresenta l'istanza della classe, che viene acceduto tramite un metodo pubblico *getInstance*. In questo modo non è possibile creare altre istanze della classe ma una sola istanza è utilizzabile alla volta. Inoltre è stata implementata anche la **lazy initialization** cioè l'istanza viene creata non quando la classe viene caricata ma alla prima richiesta. Vediamo un estratto della classe University:

```
class University {
public:
        static University *getInstance ();

private:
        static University *instance;
        University ();
        University (University const& other);
        void operator= (University const& other);
};

// Dichiaro la variabile che è stata precedentemente definita
University *University::instance = NULL;

University *University::getInstance() {
        return instance ? instance : (instance = new University());
}
```

Anche il costruttore di copia e il l'operatore = sono stati ridefiniti (e non implementati) per non permettere l'alterazione o la copia dell'istanza singleton.

4.3 Facade Pattern

La classe University è un Facade per Studente, Professore e Dottorando e garantisce infatti l'accesso a dei metodi complessi che fanno riferimento a queste classi. Il pattern è stato implementato ad esempio per i costruttori attraverso i seguenti passi:

- La classe University è **friend** di Studente, Professore e Dottorando e fornisce dei metodi per creare delle istanze di queste classi e controllarne la creazione.
- I costruttori delle classi Studente, Professore e Dottorando sono protected; inoltre i costruttori di copia e l'operatore = sono privati.

Così l'unico modo per creare delle istanze è passare dalla classe University, che ne assegna una matricola univoca e aggiunge anche l'istanza alla sua lista delle persone.

4.4 Visitor Pattern

I visitor pattern sono utilissimi per due motivi:

- Ci permettono di effettuare il **double dispatching**, e quindi di invocare metodi diversi non sono in base al tipo effettivo del chiamante ma anche a quello del chiamato.
- Ci permettono di creare classi che al loro interno contengono l'intero funzionamento di un meccanismo che cambia la sua dinamica al variare del tipo dell'oggetto.
  - In questo progetto si voleva ad esempio avere dei metodi che calcolassero in base al tipo di Persona, lo sconto da applicare sui servizi universitari quali la mensa: la classe **Discounter**.
  - Il cambio del meccanismo di calcolo comporterebbe la modifica dello stesso in ogni classe. Utilizzando i Visitor invece possiamo avere tutta la specifica nella stessa classe (l'implementazione del visitor).

Per ottenere questo funzionamento sono state definite due classi:

- Visitor (classe astratta implementata dai visitor)

```
class Visitor {
public:
        virtual ~Visitor() { }
        virtual void visit (Studente*) = 0;
        virtual void visit (Professore*) = 0;
        virtual void visit (Dottorando*) = 0;
};
```

- Visitable (classe astratta implementata da chi vuole essere visitato)

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

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

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

4.4.1 Template Pattern per specifica del tipo restituito

Per poter definire il tipo di ritorno del visitor è stato utilizzato il **Template Pattern** in modo da specificare il tipo di ritorno del Visitor. In questo modo si è creato un wrapper alla classe Visitor che contenga anche un campo *value* che contenga il valore di ritorno del visitor e possa essere acceduto da chi ha invocato il visitor sull'oggetto.

```
template <typename ReturnType>
class ReturningVisitor: public Visitor {
public:
        ReturnType const& getValue () { return value; }

protected:
        void setValue (ReturnType const& value) { this->value = value; }

private:
        ReturnType value;
};
```

4.4.2 Visitors are Welcome!

Quante volte nei film di Hollywood abbiamo visto questa frase scritta sui cartelli dei pazzi che aspettano l'arrivo degli alieni sui grattacieli di New York, e che poi vengono sistematicamente annientanti dagli alieni stessi?

Anche in questo progetto la classe Visitor è dichiarata **friend** delle classi nella gerarchia di Persona. In questo modo si può trarre vantaggio dalla potenza dei Visitor senza dover rinunciare al principio dell'incapsulamento mettendo tutti i metodi pubblici. I Visitors potranno quindi aver accesso anche ai membri privati e protetti delle classi della gerarchia di Persona.

5 Altro

Altri costrutti che sono stati utilizzati nel progetto sono:

- Costruttori di Copia
- Passaggio per valore (solitamente utilizzato per i tipi primitivi)
- Passaggio per reference (utilizzato per essere più veloce e snello, con costruttori di copia quando è stato necessario salvare una copia dell'oggetto all'interno dell'istanza oppure per proteggere e da qualificatori **const** per proteggere dalla modifica gli oggetti passati come parametri o ritornati (ad esempio per i getter).
- Passaggio per puntatore (snello ma non previene la delete dell'oggetto).

Parte III
JML

Java Modeling Language (JML) è un linguaggio che estende Java con funzionalità tipiche della programmazione Design by Contract; in particolare si possono aggiungere alle classi e ai metodi Java precondizioni, postcondizioni e invarianti atti a garantire sia a chi invoca il metodo che a chi lo implementa il rispetto di contratti scritti in fase di design. Le specifiche dei contratti sono inserite come commenti Java, in questo modo da preservare la compilazione anche con un classico compilatore Java (perdendo ovviamente i controlli relativi ai contratti).

Esistono diversi tipi di verificatori di contratti JML, ma i più famosi sono sicuramente RAC (verifica dei contratti dinamica in runtime) e ESC (verifica statica con possibilità di provare la correttezza del programma). Su questi due tipi di verifica si basano tutti i tool scritti per JML come quello usato in questo elaborato: OpenJML.

1 Descrizione del progetto

Il progetto riprende il tema dell'Università svolto per l'elaborato in C++, e in particolare la parte relativa allo studente e al salvataggio dei voti a cui sono state applicate alcune modifiche per renderla più interessante dal punto di vista di Java e di JML.

Le funzionalità che si sono volute sviluppare sono state:

- Mantenimento dell'anagrafica, dell'anno di corso e dei voti conseguiti dallo Studente con metodi per accedere a tali dati e modificarli.
- Voti salvati in una struttura che li mantenga ordinati e verifica di questa proprietà attraverso un contratto della classe.

Si faccia riferimento alla sottosezione Classi e funzionalità per i dettagli.

1.1 Obiettivi

L'obiettivo è stato quello di utilizzare tutti i costrutti disponibili in JML per scrivere contratti che garantissero la validità del codice scritto. Per fare questo l'approccio seguito è stato:

1. Formulazione del problema di carattere generale.
2. Individuazione e generalizzazione delle classi e delle strutture dati delle classi, delle funzionalità e dei metodi che astraessero i comportamenti desiderati in unità logiche a sé stanti.
3. Individuazione degli invarianti di classe e scrittura dei contratti per classi e metodi.
4. Implementazione in modo da garantire il rispetto dei contratti.

Dal mio punto di vista è stato molto importante la scrittura dei contratti in un tempo precedente alla codifica, in questo modo infatti si sono tralasciati tutti gli aspetti implementativi e ci si è concentrati solo sulla funzionalità voluta da ogni metodo.

Uno dei limiti all'approccio del Design By Contract è infatti quello della scrittura dei contratti successivamente alla scrittura dei metodi. In questo modo infatti si rischia di legare troppo la funzionalità voluta all'aspetto implementativo. È stato questo il motivo che mi ha spinto a non utilizzare codice già scritto per altri corsi per questo elaborato, ma affrontare una problematica nuova.

1.2 Classi e funzionalità

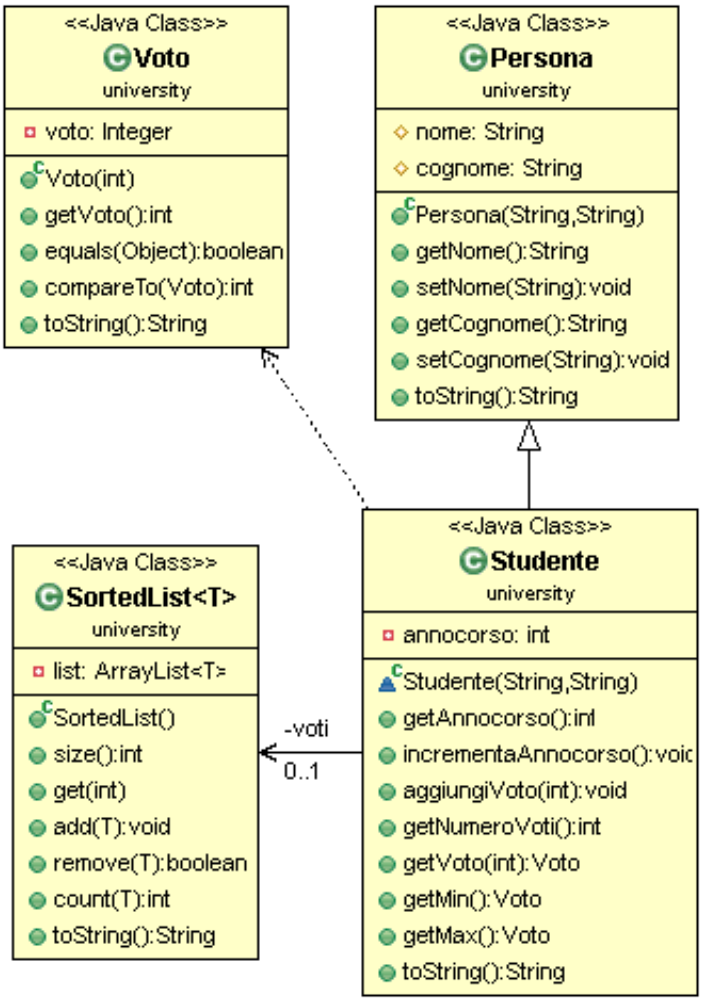
Il progetto consta di 4 classi principali:

Persona (*Persona.java*) Racchiude l'anagrafica con nome e cognome e i metodi per accedervi

Studente (*Studente.java*) Estende la classe Studente e vi aggiunge l'anno di corso nel quale lo studente si trova e la lista ordinata dei suoi voti registrati.

Voto (*Voto.java*) È un wrapper di Integer che lo rende però non modificabile. È utilizzato per rappresentare un voto nella lista dei voti dello Studente.

SortedList<T extends Comparable<? super T>> (*SortedList.java*) È un wrapper di ArrayList in modo da garantisce l'inserimento ordinato in tempo lineare *O(n)* nel caso migliore, peggiore e medio. L'accesso è effettuato in tempo costante *O(1)*. La classe è parametrica di tipo *T extends Comparable<? super T>* per garantire la possibilità di ordinare (attraverso il metodo *compareTo* dell'interfaccia *Comparable*) gli elementi della lista.



Oltre a queste 4 classi il progetto si compone di un'ulteriore classe Demo (*Demo.java*) che implementa il metodo *main* e che dimostra l'utilizzo della classe e attraverso la quale si può vedere il verificatore di contratti in runtime all'opera.

2 Costrutti JML

I contratti JML si inseriscono come commenti particolari all'interno del codice Java. Si possono usare due diverse sintassi:

/*@ ... @*/ Usata generalmente per i contratti multi-linea e per i contratti inseriti all'interno di uno statement Java (come *non_null* o *pure* che vedremo in seguito).

//@ ... Usata generalmente per i contratti che stanno su una sola riga.

2.1 Metodi puri

Vengono definiti puri i metodi che non hanno effetti collaterali, ossia che non variano lo stato dell'oggetto sul quale sono invocati. In JML la parola chiave per indicare che un metodo è puro è **pure** e deve essere messa come annotazione JML nella segnatura del metodo, dopo i qualificatori ma prima del tipo restituito. I metodi puri sono gli unici che possono essere usati in maniera sicura all'interno dei contratti JML, infatti se viene invocato un metodo non marcato come puro in un contratto JML, questo genera un warning in compilazione.

Esempio del metodo puro *size()* della classe *SortedList*:

```
public /*@ pure @*/ int size() {
        return list.size();
}
```

Nel codice si è fatta attenzione a marcare tutti i metodi senza side-effects come puri, ad esempio di hanno:

- Metodi getter
- Metodi come *count* e *size*
- Metodi *equals*, *compareTo* e *toString*

2.2 non_null

Il qualificatore JML **non_null** specifica che il campo o il parametro o il valore di ritorno di una funzione sul quale viene applicato non può essere nullo; questo viene posto tra i qualificatori Java e il tipo. Esempio da *Persona*:

```
protected /*@ non_null @*/ String nome;
protected /*@ non_null @*/ String cognome;
```

2.3 Precondizioni e Postcondizioni

Precondizioni, Postcondizione e Invarianti (che vedremo in una sezione successiva), sono le tre componenti fondamentali anche della logica di Hoare[3], infatti i programmi scritti in JML possono anche essere verificati attraverso tool come KeY-Hoare.

[3]Tony Hoare, vincitore del Turing Award e inventore dell'algoritmo Quicksort nonché della logica di Hoare

2.3.1 Precondizione

La keyword **requires** serve per specificare le precondizioni di un metodo. Queste vengono verificate prima dell'esecuzione dello stesso.

Esempio dal metodo *get(int index)* classe *SortedList* che garantisce che l'indice richiesta esista all'interno della lista:

```
/*@ requires (index >= 0)
  @         && (index < size());
  @*/
public /*@ pure @*/ T get(int index) {
        return  list .get(index);
}
```

Si noti che è possibile invocare il metodo *size()* all'interno del metodo perché questo è stato marcato e verificato puro (vedi sottosezione metodi puri).

2.3.2 Postcondizioni

La keyword **ensures** serve per specificare le postcondizioni di un metodo. Queste vengono verificate al termine dell'esecuzione dello stesso.

Esempio dal costruttore della classe Studente:

```
/*@ ensures (nome == getNome())
  @       && (cognome == getCognome())
  @       && (annocorso == 1)
  @       && (getNumeroVoti() == 0);
  @*/
Studente(/*@ non_null @*/ String nome, /*@ non_null @*/ String cognome) {
        super(nome, cognome);
        annocorso = 1;
        voti  = new SortedList<Voto>();
}
```

È utile verificare che i setter funzionino correttamente verificando ad esempio che alla fine del metodo risulti *nome == getNome()*, ossia una successiva invocazione del metodo *getNome()* restituirà il nome corretto con cui l'oggetto è stato costruito.

2.3.3 Campi privati

Se si vuole accedere a campi privati della classe dai contratti, è necessario aggiungere il qualificatore JML **spec_public** dopo i qualificatori Java del campo e prima del tipo di questo.

Esempio del campo *annocorso*:

```
private /*@ spec_public @*/ int annocorso;
```

2.3.4 Valore di ritorno

È possibile utilizzare il valore di ritorno del metodo nelle postcondizioni utilizzando la keyword **\result**.

Esempio da *Persona*, in questo caso si vuole controllare che la stringa restituita dal metodo *toString()* sia composta da almeno 3 caratteri (un carattere per il nome, uno per lo spazio e uno per il cognome):

```
//@ ensures (\result).length() >= 3;
@Override
public /*@ pure @*/ /*@ non_null @*/ String toString() {
        return nome + " " + cognome;
}
```

Si noti che il qualificatore *@Override* non fa parte della specifica JML ma Java. La specifica JML va posta precedentemente anche ai qualificatori Java.

2.3.5 Valori precedenti alla chiamata

Nelle postcondizioni è possibile utilizzare il valore di un campo o addirittura quello di un metodo precedentemente all'invocazione del metodo stesso. Questo effetto viene ottenuto utilizzando la keyword **\old(*value*)**.

Esempio di utilizzo nella classe *Studente* sul campo *annocorso*:

```
//@ ensures this.annocorso == \old(annocorso) + 1;
public void incrementaAnnocorso() {
        ++annocorso;
}
```

Esempio di utilizzo nella classe *SortedList* su dei metodi (questo metodo verifica che l'inserimento sia ordinato, ma non viene controllata la postcondizione sull'ordinamento perché è già controllata dall'invariante della classe, si veda la sezione Invarianti):

```
/*@ ensures (size() == (\old(size()) + 1))
  @       && (count(element) == (\old(count(element)) + 1));
  @ */
public void add(/*@ non_null @*/ T element) {
        int i = 0;
        while (i < list.size() && list.get(i).compareTo(element) <= 0) { ++i; }
        list.add(i, element);
}
```

2.4 Contratti delle sottoclassi

Quando viene fatto override di un metodo in una sottoclasse, è possibile unire i contratti del metodo nella superclasse con quelli del metodo nella sottoclasse con la keyword **also**. Esempio del metodo *toString()* di *Studente* che estende *Persona*:

```
/*@ also
  @ ensures (\result).length() > 20;
  @ */
@Override
public /*@ pure @*/ /*@ non_null @*/String toString() {
        return nome + " " + cognome + ", " + annocorso + " anno, voti: " + voti.toString();
}
```

2.5 Implicazione

Vi sono 4 tipi di implicazione che si possono inserire nei contratti JML:

- ==> (implica)

  Utilizzato insieme ai quantificatori universali e esistenziali, si veda la sezione dedicata.

- <== (segue)

  Utilizzato per rendere più chiari i metodi *equals*.

```
//@ ensures (voto == ((Voto)obj).voto) <== (\result == true);
@Override
public /*@ pure @*/ boolean equals(Object obj) {
        if (!(obj instanceof Voto)) return false;
        return voto.equals(((Voto)obj).voto);
}
```

- <==> (se e solo se)

  Utilissimo in diversi casi, come nei metodi *compareTo*.

```
/*@ ensures (voto <= other.voto) <==> (\result <= 0)
  @       && (voto >= other.voto) <==> (\result >= 0);
  @*/
@Override
public /*@ pure @*/ int compareTo(/*@ non_null @*/ Voto other) {
        return voto.compareTo(other.getVoto());
}
```

- <=!=> (non (se e solo se))

  Questa implicazione è stata evitata perché rende difficile la lettura dei contratti.

2.6 Qualificatore universale e esistenziale

JML permette l'uso dei qualificatori universale (∀) e esistenziale (∃), rispettivamente con le keywords **\forall** e **\exists**. Questo semplifica notevolmente la scrittura dei contratti.

Esempio dal metodo *getMax()* di Studente che ritorna il voto più alto nella *SortedList* di Studente:

```
/*@ ensures (\forall int i; i >= 0 && i < getNumeroVoti() ==> (\result).compareTo(voti.get(i)) >= 0)
  @       && (\exists int i; i >= 0 && i < getNumeroVoti() ==> (\result).equals(voti.get(i)));
  @*/
public /*@ pure @*/ Voto getMax() {
        return voti.get(voti.size() - 1); // La lista è ordinata in maniera non decrescente.
}
```

2.7 assert

Gli assert permettono di inserire dei controlli di condizioni in punti all'interno del codice. Questa funzionalità, disponibile anche in Java, si usa con la keyword **assert**. Tuttavia nel codice non sono state usate volutamente. Infatti in un linguaggio di Design by Contract a mio parere la presenza di assert sottolinea dei problemi di design, probabilmente il fatto che il metodo fa più di una cosa, e quindi si consiglia il re-design del metodo, estraendone e separandone le funzionalità differenti per poter poi mettere pre/postcondizioni.

Se proprio risultasse necessario utilizzare delle assert, quelle di JML sono più espressive, infatti permettono l'utilizzo dei qualificatori universale e esistenziale.

2.8 Invarianti

Viene detto invariante una condizione che deve valere prima e dopo ogni invocazione di un metodo sull'oggetto in questione. Gli invarianti sono molto utili, infatti ci permettono di sintetizzare il contratto di una intera classe e dei suoi metodi in un unico punto, evitando di doverlo ripetere nelle pre/postcondizioni dei singoli metodi.

JML ci permette di specificare gli invarianti con la keyword ***invariant*** che può essere anche preceduta da ***private*** se l'invariante fa riferimento a campi privati della classe.

- *Persona* impone che non siano validi nomi e cognomi vuoti:

```
public class Persona {
        /*@ private invariant nome != ""
          @       && cognome != "";
          @*/
   ...
}
```

- *Studente* impone che l'anno di corso sia sempre maggiore di 0:

```
public class Studente extends Persona {
        //@ private invariant (annocorso >= 1);
   ...
}
```

- *Voto* impone quali siano i voti validi. Si noti che questo è l'unico contratto che descrive quali siano i voti validi, quindi in caso si volesse cambiare il meccanismo dei voti e permettere i voti fino al 32, basterebbe cambiare questo contratto (e l'implementazione di conseguenza).

```
public class Voto implements Comparable<Voto> {
        //@ private invariant (voto >= 18) && (voto <= 30);
   ...
}
```

- *SortedList* impone che la lista sia ordinata in maniera non decrescente. Anche qui si noti che questo è l'unico contratto che descrive che la classe rappresenta una lista ordinata.

```
public class SortedList<T extends Comparable<? super T>> {
        //@ invariant (\forall int i; (i >= 0 && i < size()); get(i).compareTo(get(i+1)) <= 0);
   ...
}
```

3 Demo con tracce di esecuzione con violazione dei contratti

```
package university;
public class Demo {
        public static void main(String[] args) {

                Studente enrico = new Studente("Enrico", "Bacis");
                // ERROR:
                // Studente enrico = new Studente("", "Bacis");
                // Studente enrico = new Studente("Enrico", "");

                // Inserisco 4 voti
                enrico.aggiungiVoto(30);
                enrico.aggiungiVoto(27);
                enrico.aggiungiVoto(18);
                enrico.aggiungiVoto(28);
                // ERROR:
                // enrico.aggiungiVoto(17);
                // enrico.aggiungiVoto(31);

                // Verifichiamo che la lista sia ordinata
                System.out.println("Voti Inseriti : " + enrico.getNumeroVoti());
                System.out.println(enrico);

                // Modifico l'anno di corso
                System.out.println();
                enrico.incrementaAnnocorso();
                System.out.println("Anno di corso: " + enrico.getAnnocorso());

                // Verifica di max, min, numerovoti
                System.out.println();
                System.out.println("Voto più basso: " + enrico.getMin());
                System.out.println("Secondo voto più basso: " + enrico.getVoto(1));
                System.out.println("Secondo voto più alto:  "
                                + enrico.getVoto(enrico.getNumeroVoti() - 2));
                System.out.println("Voto più alto:  " + enrico.getMax());
                //ERROR:
                // System.out.println("Voto inesistente: " + enrico.getVoto(-1));
                // System.out.println("Voto inesistente: "
                                + enrico.getVoto(enrico.getNumeroVoti()));

                // Cambio di identità
                enrico.setNome("Buffer");
                enrico.setCognome("Overflow");
                // ERROR:
                // enrico.setNome("");
                // enrico.setCognome("");
                System.out.println();
                System.out.println(enrico);
        }
}
```

4 Supporto allo sviluppo [MAKE]

Volendo l'utilizzo del tool make (http://en.wikipedia.org/wiki/Make_(software)) da un po' di tempo, ho deciso di iniziare scrivendo un Makefile di supporto allo sviluppo di codice JML utilizzando la libreria OpenJML.

```
# Customize this part on your setting


SRCDIR     = src
OPENJMLLIB = lib/openjml.jar
PACKAGE = university
MAINCLASS = Demo


# Customize this is you are not on Linux/Mac
# (a better way would be to switch to Linux/Mac)


SEP  = /
CPSEP = :


# Do not change after this point unless you know what you are doing


ifeq ($(SRCDIR),)
  SRCDIR = .
endif

ifeq ($(PACKAGE),)
  MAIN = $(MAINCLASS)
else
  MAIN = $(PACKAGE).$(MAINCLASS)
endif


SOURCES = $(wildcard $(addprefix $(SRCDIR)$(SEP)$(PACKAGE)$(SEP)*, java))
CLASSES = $(addsuffix .class, $(basename $(SOURCES)))
OPENJML = $(CURDIR)$(SEP)$(OPENJMLLIB)


run: all


all: clean clearscr rac exec


rac: clearscr
	@echo 'Compiling with OpenJML (RAC)...'
	@echo ''
	@java -jar "$(OPENJML)" -rac -noPurityCheck $(SOURCES)
```

```
exec:
	@echo ''
	@echo 'Executing with OpenJML...'
	@echo ''
	@cd $(SRCDIR) && java -cp "$(OPENJML)$(CPSEP)." $(MAIN)


check: clearscr
	@echo 'Checking with OpenJML ...'
	@echo ''
	@java -jar "$(OPENJML)" -check -noPurityCheck $(SOURCES)


check-purity: clearscr
	@echo 'Checking with OpenJML (Purity enabled with hidden openjml.jar errors) ...'
	@echo ''
	@java -jar "$(OPENJML)" -check $(SOURCES) | awk "/^$(SRCDIR)/,/ +\^$$/"


check-purity-all: clearscr
	@echo 'Checking with OpenJML (Purity enabled) ...'
	@echo ''
	@java -jar "$(OPENJML)" -check $(SOURCES)


clean:
	@rm -rf $(CLASSES)
	@echo 'Done Cleaning'
	@echo ''


clearscr:
	@clear


help:
	@echo 'Usage: make [TARGET]'
	@echo 'TARGETS:'
	@echo '  all                (=make) compile and execute with OpenJML (RAC).'
	@echo '  rac                compile only with OpenJML (RAC).'
	@echo '  exec               execute with OpenJML without recompiling.'
	@echo '  check              check with OpenJML.'
	@echo '  check-purity       check with OpenJML and Purity (lib warnings disabled).'
	@echo '  check-purity-all   check with OpenJML and Purity (lib warnings enabled).'
	@echo '  clean              clean class files .'
	@echo '  help               print this message.'
```

Il target di default invocato con il comando *make* si occupa di compilare il codice presente nella *SRCDIR* con il RAC di OpenJML e di eseguirlo attivando i controlli in runtime.

Parte IV
Abstract State Machines

1 Descrizione del progetto

In primo luogo l'idea era stata quella di dare una rappresentazione ASM della macchina a stati utilizzata da Martin Fowler[4] nel libro UML Distilled: un sistema di sicurezza a cui fa capo un coniglio mannaro (tanto caro al Prof. Salvaneschi).

Tuttavia questa non poneva delle grandi difficoltà dato che gli stati erano minimali, così come le informazioni da salvare. Si è quindi deciso di analizzare la procedura che controlla la registrazione dei voti di una Università, per riprendere il tema dei progetti di C++ e di JML. In questo caso ci si è focalizzati più sull'aspetto di sicurezza, quindi permettendo solamente ai professori registrati nel sistema di inserire voti per gli Studenti.

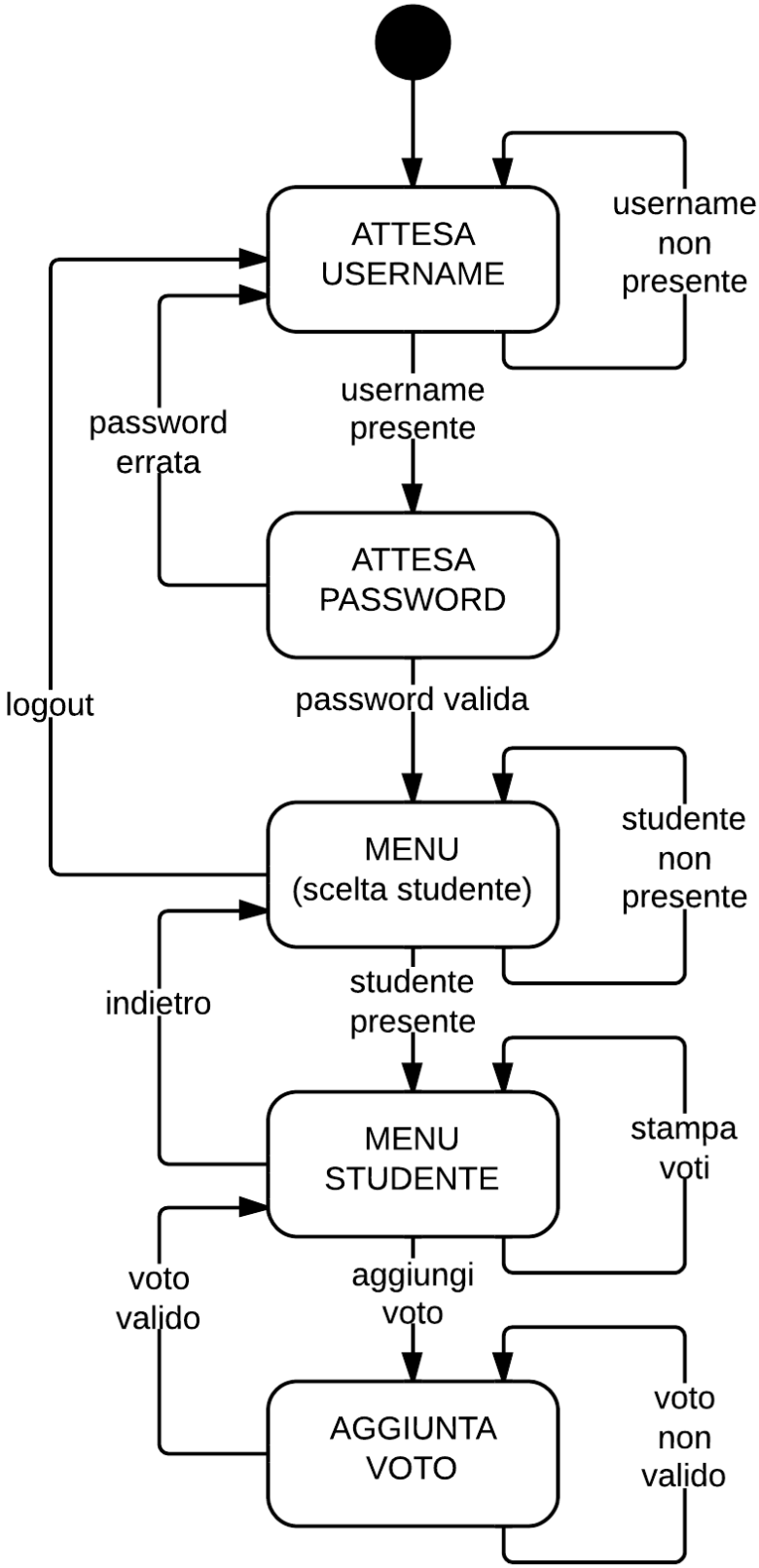
Il progetto deve rispettare le seguenti specifiche:

- Registrazione da parte dei professori di voti (validi) nella lista di voti degli studenti registrati.
- Assicurazione che solamente i professori abbiano accesso al menu per l'inserimento dei voti.
- Possibilità di effettuare il logout da parte del Professore.
- Registrazione in sequenza ordinata in maniera temporale dei voti per lo studente.
- Visualizzazione di tutti i voti dello studente.

[4] http://martinfowler.com/books/uml.html

1.1 Macchina a stati

In base alle specifiche si è dapprima scritta la macchina a stati del progetto. Il formato è quello standard per le macchine a stati descritte attraverso UML.



2 Standard Library

Si è fatto uso della Standard Library soprattutto per quanto riguarda l'utilizzo di Sequence, String, e funzioni su di esse come toString, concat per le stringhe, or per i valori Booleani. Si veda il codice per maggiori dettagli.

3 Domini

```
domain Voto subsetof Integer
abstract domain Professore
dynamic abstract domain Studente

enum domain Stati = { ATTESA_USERNAME | ATTESA_PASSWORD | MENU
                    | ATTESA_STUDENTE | MENU_STUDENTE | AGGIUNTA_VOTO }
enum domain Comandi = { SCELTA_STUDENTE | LOGOUT }
enum domain ComandiStudente = { AGGIUNGI_VOTO | STAMPA_VOTI | INDIETRO }
```

- Voto è un subset di Integer, successivamente verrà meglio caratterizzato per i valori che può assumere.
- Professore e Studente sono degli abstract domain, sono come delle classi che non sono subset di altre classi.
- Studente è una classe dinamica perché lascia spazio all'inserimento di nuovi studenti oltre a quelli definiti nel file.
- Stati, Comandi e ComandiStudente sono degli enum e rappresentano gli stati della macchina a stati il primo, i comandi disponibili dal menù il secondo e i comandi disponibili dal menù studente il terzo.

4 Funzioni

4.1 Funzioni dinamiche

La keyword dynamic indica che il valore della funzione può cambiare valore con il tempo, quindi è una variabile.

4.1.1 Controllate

Sono le funzioni che possono essere lette e scritte solo dalla macchina a stati.

```
dynamic controlled outMsg: Any
dynamic controlled statoSistema: Stati
dynamic controlled professoreCorrente: Professore
dynamic controlled studenteCorrente: Studente
dynamic controlled getVoti: Studente -> Seq(Voto)
```

Any indica che *outMsg* (che è il messaggio di output del sistema) può assumere valori di qualsiasi dominio.

4.1.2 Monitorate

Queste funzioni possono essere scritte sono dall'environment, ma possono essere anche lette dalla macchina a stati.

```
dynamic monitored comando: Comandi
dynamic monitored comandoStudente: ComandiStudente
dynamic monitored professore: Professore
dynamic monitored studente: Studente
dynamic monitored password: String
dynamic monitored nome: String
dynamic monitored voto: Voto
```

Si usano questo tipo di variabili per inserire comandi e valori nella macchina a stati dall'esterno. È possibile utilizzare sia un file che la console per l'inserimento di questi valori.

5 Funzioni statiche

5.1 Costanti

Queste variabili non possono mai cambiare valore.

```
static gargantini: Professore
static scandurra: Professore
static enrico: Studente
static secondo: Studente
static terzo: Studente
static quarto: Studente
static debugFlag: Boolean
```

5.2 Derivate

Queste costanti derivano da funzioni che assegnano staticamente i loro valori ma possono variare per via degli argomenti. Sono come delle tabelle con dei valori che non si possono cambiare.

```
derived getPassword: Professore -> String
derived getNome: Professore -> String
derived getNome: Studente -> String
```

Le funzioni dalle quali derivano si vedranno tra poco.

6 Funzioni

Qui vengono definiti i domini, le costanti e le funzioni.

- Voto può assumere solo valori interi compresi tra 18 e 30.
- *debugFlag* può valere *true* o *false*. Se è *true* permette l'accesso anche con password errata. Questo è utile per poter utilizzare il run randomly, che altrimenti non indovinerebbe mai le password.
- le funzioni vengono definite attraverso degli *switch* sull'oggetto che viene passato come parametro.

```
domain Voto = {18..30}

function debugFlag = false

function getPassword($p in Professore) =
    switch($p)
        case gargantini : "johnmitchell"
        case scandurra  : "iloveasmeta"
    endswitch

function getNome($p in Professore) =
    switch($p)
        case gargantini : "Angelo Gargantini"
        case scandurra  : "Patrizia Scandurra"
    endswitch

function getNome($s in Studente) =
    switch($s)
        case enrico   : "Enrico Bacis"
        case secondo : "Secondo"
        case terzo    : "Terzo"
        case quarto   : "Quarto"
    endswitch
```

7 Regole

7.1 Stampa Voti

```
macro rule r_stampaVoti( $s in Studente ) =
    outMsg := toString( getVoti( $s ) )
```

Stampa i voti contenuti nella sequenza di voti dello studente.
Si noti l'utilizzo di:

macro rule Si utilizza per definire delle regole Macro (in questo caso senza parametri).

:= Si utilizza per fare l'update di una funzione dinamica.

7.2 Attesa Username

```
macro rule r_attesaUsername =
    if ( statoSistema = ATTESA_USERNAME ) then
        if ( exist unique $p in Professore with $p = professore ) then
            par
                professoreCorrente := professore
                statoSistema := ATTESA_PASSWORD
                outMsg := "Inserire password"
            endpar
        endif
    endif
```

Questa regola viene eseguita quando la macchina a stati è in attesa dello username. Effettua un controllo sul fatto che la variabile monitorata *professore* inserita dall'esterno sia effettivamente un Professore; se così non è, non si procede all'inserimento della Password.

if [else] endif Si utilizza per inserire blocchi condizionali.

exist[unique] with Si utilizza per verificare che esista una (o anche uno sola) entità nel dominio che soddisfi la condizione *with*.

par endpar Si utilizza per inserire operazioni in parallelo.

7.3 Attesa Password

```
macro rule r_attesaPassword =
    if ( statoSistema = ATTESA_PASSWORD ) then
        let ( $password = getPassword(professoreCorrente) ) in
            if ( or( debugFlag, $password = password ) ) then
                par
                    statoSistema := MENU
                    outMsg := concat( "Buongiorno Prof. ", getNome( professoreCorrente ) )
                endpar
            else
                par
                    statoSistema := ATTESA_USERNAME
                    outMsg := "Accesso negato"
                endpar
            endif
        endlet
    endif
```

Controlla che la password relativa al Professore selezionato sia quella corretta. Se è così porta al menù, altrimenti rimanda alla schermata di inserimento dello username.

7.4 Menu

```
macro rule r_menu =
    if ( statoSistema = MENU ) then
        par
            if ( comando = SCELTA_STUDENTE ) then
                par
                    statoSistema := ATTESA_STUDENTE
                    outMsg := "Inserire lo studente"
                endpar
            endif
            if ( comando = LOGOUT ) then
                par
                    statoSistema := ATTESA_USERNAME
                    outMsg := "Arrivederci"
                endpar
            endif
        endpar
    endif
```

Da la possibilità al Professore di procedere con la scelta di uno studente da modificare oppure effettuare il logout e tornare alla schermata di inserimento username.

7.5 Scelta Studente

```
macro rule r_sceltaStudente =
    if ( statoSistema = ATTESA_STUDENTE ) then
        if ( exist unique $s in Studente with $s = studente ) then
            seq
                studenteCorrente := studente
                statoSistema := MENU_STUDENTE
                outMsg := concat("Menu: ",getNome(studenteCorrente))
            endseq
        endif
    endif
```

Da la possibilità di scegliere lo studente da modificare. Anche qui viene controllato che lo studente sia effettivamente uno studente del dominio degli studenti.

Si noti l'utilizzo di:

seq endseq Si utilizza per inserire operazioni in sequenza. Questo è utilizzato perché è importante in questo caso che lo *studenteCorrente* venga aggiornato prima di modificare *outMsg* in cui viene utilizzato. Si sarebbe potuto anche mettere *studente*_per ovviare al problema.

7.6 Menu Studente

```
macro rule r_menuStudente =
    if ( statoSistema = MENU_STUDENTE ) then
        par
            if ( comandoStudente = AGGIUNGI_VOTO ) then
                par
                    statoSistema := AGGIUNTA_VOTO
                    outMsg := "Inserire il voto"
                endpar
            endif
            if ( comandoStudente = STAMPA_VOTI ) then
                r_stampaVoti[ studenteCorrente ]
            endif
            if ( comandoStudente = INDIETRO ) then
                par
                    statoSistema := MENU
                    outMsg := "Menu principale"
                endpar
            endif
        endpar
    endif
```

Permette al Professore di selezionare se aggiungere un voto, stampare i voti correnti o tornare al menu principale.

7.7 Aggiunta Voto

```
macro rule r_aggiuntaVoto =
    if ( statoSistema = AGGIUNTA_VOTO ) then
        if ( exist $v in Voto with $v = voto ) then
            par
                getVoti( studenteCorrente ) :=
                    append( getVoti( studenteCorrente ), voto )
                outMsg := concat( "Voto inserito: ", toString( voto ) )
                statoSistema := MENU_STUDENTE
            endpar
        endif
    endif
```

Se il voto inserito dall'utente è un voto valido nel dominio Voto, allora questo viene accodato alla sequenza dei voti dello studente.

8 Main

```
main rule r_Main =
seq
    r_attesaUsername[]
    par
        r_attesaPassword[]
        r_menu[]
        r_sceltaStudente[]
        r_menuStudente[]
        r_aggiuntaVoto[]
    endpar
endseq
```

La regola principale da cui viene eseguito il programma. Questa si limita a chiamare le altre regole degli stati, anche se, grazie alla variabile *statoSistema*, solo una regola verrà attivata per volta.

9 Stato Iniziale

```
default init initial_state :

    function statoSistema = ATTESA_USERNAME
    function outMsg = "Inserire username"
    function getVoti( $s in Studente ) = []
```

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

- Lo stato *ATTESA_ USERNAME*
- Un messaggio da stampare ad output
- Una sequenza vuota che rappresenta i voti per ogni Studente nel dominio degli Studenti.

Parte V
Python e Ruby

1 Python

Python è un linguaggio dal mio punto di vista fantastico che penso sia fondamentale imparare in un percorso di studi informatico perché apre la mente e aiuta il programmatore ad affacciarsi ai problemi con una rinnovata motivazione e uno sguardo diverso.

La filosofia di Python è quella di migliorare la leggibilità del codice, quindi utilizza una sintassi chiara ed espressiva. Python supporta diversi paradigmi di programmazione come OOP, programmazione imperativa e funzionale. È un linguaggio dinamicamente tipato, questo significa che non si specificano i tipi delle variabili, ma viene fatta inferenza del tipo quando si definisce la variabile, inoltre è più corretto chiamare queste entità "nomi" o "alias", infatti i nomi possono essere legati a tipi diversi di variabile durante il codice con un semplice assegnamento.

L'interprete di Python è disponibile per diversi sistemi operativi e l'implementazione più utilizzata è CPython, che segue un modello di sviluppo community-based grazie alle PEP[5], che sono proposte alla community dagli sviluppatori, discusse e in caso integrate nel linguaggio. Altri interpreti famosi sono Jython (basato su JVM) e IronPython (basato su framework .NET)

1.1 La filosofia

La filosofia con cui scrivere i programmi è egregiamente espressa nella PEP20 (the Zen of Python) che si può leggere aprendo un qualsiasi interprete di Python e scrivendo: *import this*

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!
```

1.2 Storia

Python è stato ideato da Guido van Rossum alla fine degli anni novanta come successore del linguaggio ABC (che a sua volta era stato ispirato dal linguaggio SETL del 1969). Il nome fu

[5]Python Enhancement Proposal

scelto per via della passione di van Rossum per i Monty Python e per la loro serie televisiva.

Python 2.0 è stato rilasciato nel 2000, con l'aggiunta di funzionalità come il garbage collector e il supporto per Unicode; inoltre in questo momento si ha avuto il passaggio al community-driven development di Python.

Python 3.0 è stato rilasciato alla fine del 2008 e non è retrocompatibile con le versioni precedenti; un esempio banale è quello della divisione, infatti fino a Python 2.0 a/b tra due numeri interi dava come risultato un intero (come nella migliore tradizione), ma in Python 3.0 si è finalmente andati oltre questa tradizione, come è giusto che sia per un linguaggio dinamicamente tipato dove il dato è più importante del suo contenitore, e quindi a/b anche tra interi può restituire un numero in virgola mobile (per ottenere la divisione intera si usa ora $a//b$).

Python ha vinto per 2 anni il premio TIOBE per il linguaggio di programmazione dell'anno (nel 2007 e nel 2010).

1.3 Confronto con altri linguaggi visti

È molto interessante confrontare Python con i linguaggi di programmazione e le caratteristiche di funzionamento dei linguaggi analizzati durante il corso.

- Per prima cosa si ricordi che Python (nel suo utilizzo classico) è un linguaggio interpretato. Non vi è quindi distinzione tra *compile-time* e *run-time*, quindi si eliminano tutte le problematiche relative a:
  - Early Binding
  - Late Binding
- Come secondo aspetto, la sua caratteristica di essere dinamicamente tipato elide i problemi che si possono presentare quando il tipo del riferimento è diverso dal tipo dell'oggetto e quindi non vi sono le problematiche relative a:
  - Single e Double Dispatching
  - Dynamic Binding
  - Non è inoltre possibile fare overloading di metodi con stesso numero di parametri ma differenti tipi (per il fatto che il concetto di tipo non è legato alla variabile)
- Non ci sono puntatori e le strutture dati possono essere visitate solamente attraverso gli iteratori.
- Esiste un solo tipo di passaggio di variabile che è quello per reference. Se il tipo passato è mutable come le liste, allora potrà essere modificato dalla funzione, se è immutable come le tuple no. Inoltre riassegnando il nome ad un'altra variabile all'interno di una funzione, questo non modificherà il valore al di fuori di tale funzione (all'esterno il nome continua a puntare dove ha sempre puntato).
- C'è un garbage collector che si occupa di pulire tutti gli oggetti che non sono più referenziati da alcun nome.
- Tutti gli oggetti di Python sono salvati sullo heap, non è quindi possibile avere dei dangling pointers.
- I blocchi non sono espressi da parentesi graffe ma solo attraverso l'indentazione.

- È permessa l'ereditarietà multipla.
- In genere i programmi vengono scritti partendo da questa base:

```
main():
        # codice

if __name__ == '__main__':
        main()
```

Questo perché i sorgenti Python possono essere importati, con la direttiva *import* all'interno di altri sorgenti, e in questo modo si differenzia il comportamento in base al fatto che il codice venga eseguito a partire da questo file oppure solo importato in un altro (in tal caso non viene chiamata la funzione *main()*).

Nonostante questo per questa trattazione verrà spesso utilizzato semplicemente il codice.

1.4 Esempi

Ho avuto il piacere di seguire due corsi online di Python sulla piattaforma Udacity, il primo era un corso di introduzione alla programmazione con Python, che mi ha convinto sempre più che questo debba essere il linguaggio insegnato in Informatica I al posto di C, almeno per i corsi di laurea che non sono Ingegneria Informatica; il secondo corso è stato il migliore corso online che io abbia avuto il piacere di frequentare, e che consiglio a tutti: *Design of Computer Programs*[6] tenuto da Peter Norvig[7].

In questa parte, piuttosto che focalizzarsi su un progetto unico in cui non verrebbero messi in rilievo le cose più interessanti, verranno mostrati alcuni costrutti e funzionalità interessanti di Python.

1.4.1 Passaggio di funzioni

In Python le funzioni sono membri di prim'ordine, possono essere ad esempio passate come parametro per altre funzioni. Vediamo l'esempio della funzione max, a cui viene passata la funzione di *peso* che si occupa di dare un peso agli elementi per riuscire a trovarne il massimo che desideriamo; in questo caso vogliamo ottenere ad esempio la stringa più lunga, per farlo utilizziamo come funzione di peso la funzione *len* che può essere utilizzata su tutti gli oggetti iterabili per trovarne la dimensione.

```
stringhe = ["a", "bb", "ccc", "dddd", "z"]
print max(stringhe, key=len)
```

Questo stampa a video *dddd* che è la stringa più lunga.

1.4.2 Lambda Functions

Possiamo anche definire della funzioni veloci senza nome dette lambda[8], e volendo assegnare anche dei nomi temporanei a queste funzioni. Si voglia adesso sommare il peso delle lettere nelle stringhe e trovare il massimo in base a questo:

```
stringhe = ["a", "bb", "ccc", "dddd", "z"]
fun = lambda stringa: sum(ord(c) - 96 for c in stringa.lower())
print max(stringhe, key=fun)
```

[6] https://www.udacity.com/course/cs212
[7] http://en.wikipedia.org/wiki/Peter_Norvig
[8] http://it.wikipedia.org/wiki/Lambda_calcolo

Questo stampa *z* che è la stringa che la cui somma dei caratteri è maggiore. La lambda ha come parametri le variabili espresse prima del simbolo due punti, e con quelle si può operare per ritornare un risultato.

Si noti inoltre che in Python non è necessario utilizzare la keyword **return** infatti l'ultimo statement eseguito all'interno di un blocco è automaticamente il valore di ritorno di quel blocco.

1.4.3 Paradigma MapReduce

Un indispensabile paradigma di programmazione è il MapReduce[9] che si compone di:

map una funzione che mappa i valori di una lista in degli altri, come possono essere le funzioni di peso

```
stringhe = ["a", "bb", "ccc", "dddd", "z"]
print map(lambda stringa: sum(ord(c) - 96 for c in stringa.lower()), stringhe)
```

reduce si occupa di unire poi i risultati di questa lista nel seguente modo: *reduce(f, [a, b, c]) == f(f(a, b), c)*

Si voglia ora ottenere il prodotto dei pesi sopra

```
stringhe = ["a", "bb", "ccc", "dddd", "z"]
mapfun = lambda stringa: sum(ord(c) - 96 for c in stringa.lower())
redfun = lambda x, y: x * y
print reduce(redfun, map(mapfun, stringhe))
```

1.4.4 Higher Order Functions, closures, caching e ritorno di funzioni

Essendo le funzioni first class members, possono anche essere definite all'interno di altre funzioni, e anche ritornate. Il vantaggio di fare questo deriva dalle closures, infatti quando si esegue questo passaggio, la funzione si assicura che quando verrà invocata, avrà accesso a tutte le variabili a cui aveva accesso quando è stata invocata, e anche con il loro valore. Questo concetto si chiama **chiusura funzionale**.

Vediamo un esempio di funzione che wrappa una funzione che gli viene passata come parametro in un'altra che fa caching dei risultati. Questo concetto è **fondamentale** in programmazione funzionale, dove, se non si ha caching si rischia di ripetere i calcoli moltissime volte. Esemplifichiamo questo concetto con l'esempio del calcolo dell'n-esimo numero della sequenza di Fibonacci:

```
def cache(fn):
    dict = {}
    def _f(x):
        if x not in dict:
            dict[x] = fn(x)
        return dict[x]
    _f.dict = dict
    return _f

def fatt(n):
    return 1 if (n == 0) else n * fatt(n-1)

fatt = cache(fatt)
print fatt(10)
```

[9] http://en.wikipedia.org/wiki/MapReduce

In questo codice sono stati utilizzati anche dizionari e si noti inoltre che vengono assegnati a una funzione dei metodi! Possiamo infatti vedere l'intera cache della funzione con:

```
print fatt.__dict__
```

Decorator Pattern Si noti che è stato utilizzato un decorator pattern su *fatt*, che è stata wrappata (o decorata) da un'altra funzione. Questo concetto è talmente fondamentale in Python che ha una sintassi specifica, avremmo potuto infatti usare:

```
@cache          # fatt = cache(fatt)
def fatt(n):
    return 1 if (n == 0) else n * fatt(n-1)
```

1.4.5 yield e generatori

Si possono creare dei generatori che generano dei dati in maniera lazy ogni volta che viene invocato il comando next su di essi (si noti che le funzioni come map, sum e tutte quelle che utilizzano iteratori utilizzano proprio questo comando). Vediamo un generatore di numeri di Fibonacci:

```
def fib_range():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

La parola chiave **yield** indica che il flusso della funzione va in pausa in quel punto generando il dato, quando verrà riesumata ripartirà da questo punto e, grazie alla chiusura funzionale, utilizzerà gli stessi dati, come se non fosse mai stata fermata.

2 Ruby

Ruby è, tra i linguaggi Object Oriented che conosco, quello che più si distacca dagli altri, offrendo al programmatore una elasticità espressiva e una gamma di costrutti difficile da trovare in altri linguaggi. Anche in questo caso ho avuto il piacere di frequentare il corso online dell'università di Berkley dedicato a Ruby e a Ruby on Rails, di seguito verranno quindi presentati gli aspetti più interessanti che ho trovato in Ruby grazie a questo corso.

2.1 Storia

Ruby è un linguaggio interpretato completamente a oggetti. Il progetto è nato nel 1993 come progetto personale del giapponese Yukihiro "Matz" Matsumoto, Ruby è stato il primo linguaggio di programmazione sviluppato in Oriente a guadagnare abbastanza popolarità da superare la barriera linguistica che separa l'informatica nipponica da quella internazionale.

Il linguaggio che ha maggiormente ispirato l'autore è lo Smalltalk, da cui Ruby ha tratto la maggior parte delle sue caratteristiche. A seguire ci sono il Lisp (ed in generale i linguaggi funzionali), da cui provengono le chiusure (blocchi o proc, in Ruby), e il Perl, per la sintassi e l'espressività. Negli ultimi anni la popolarità di Ruby ha subito una forte impennata, dovuta alla comparsa di framework di successo per lo sviluppo di applicazioni web, come Ruby On Rails.

2.1.1 Versioni

La prima versione pubblica di Ruby è stata annunciata su un newsgrup giapponese il a dicembre del 1995 e già a questo stadio dello sviluppo erano presenti molti degli aspetti di Ruby come il design Object Oriented, i mixin, gli iteratori, le closures, la gestione delle eccezioni e il garbage collector.

Si ha poi avuto un costante sviluppo fino ad arrivare alla versione 1.8 nell'agosto del 2003. Questa versione, che ora è stata deprecata perché non più compatibile con Ruby 1.9, è stata stabile per lungo tempo e largamente utilizzata; è su questa versione che è stato sviluppato il framework Ruby on Rails (approdato nel 2005) che ha reso famoso Ruby.

Nel 2007 è stata rilasciata la versione 1.9, non retrocompatibile con la 1.8. Questa versione ha aggiunto tra le altre cose una sintassi per le lambda-functions.

Per febbraio 2013, a giorni quindi dal momento in cui questo testo viene scritto, verrà rilasciata la versione stabile di Ruby 2.0, che è stata annunciata essere completamente retrocompatibile con la versione 1.9.

2.1.2 Implementazioni dell'interprete

L'interprete ufficiale di Ruby è *Matz's Ruby Interpreter* anche noto semplicemente come *MRI*. Questa implementazione p scritta in C e utilizza una virtual machine ad-hoc per Ruby. Esistono tuttavia altre implementazioni. Le più note sono:

JRuby implementazione Java che gira su JVM

Rubinius Virtual Machine scritta in bytecode C++ che utilizza LLVM per compilare in linguaggio macchina il codice a runtime. Il compilatore bytecode e molte classi principali sono scritte in Ruby.

IronRuby Implementation che sfrutta il framework .NET

2.2 Confronto con altri linguaggi

Il confronto è molto simile a quello visto per Python, quindi si trattano qui le principali differenze proprio da Python

- Le stringhe sono mutable
- Si possono creare costanti (non si usa la keyword **const** ma semplicemente si usa la prima lettera maiuscola)
- C'è solo un tipo di container, Array, ed è mutable.
- Tutto è un oggetto, non esistono tipi primitivi, anche scrivendo 5 è un oggetto! Questo ci permette di scrivere dei cicli for semplicemente utilizzando metodi come *times* o *each*

```
5.times { print "Ciao!" }
```

- I blocchi sono delimitati da *{...}* (generalmente usato quando i blocchi sono su una riga sola) o da *do...end* (generalmente usato quando i blocchi sono su più righe).
- Non si accede mai a degli attributi direttamente, tutto è un metodo, inoltre con Syntactic Sugar, spesso non ce ne si accorge perché la parentesi per le chiamate dei metodi sono spesso omesse.
- Esistono le keyword **private** e **protected** al posto della convenzione di nomi di Python.
- Non c'è eredità multipla, si usano i mixin, che sono simili alle interfacce perché generalmente si scrivono per modellare *comportamenti* ma possono portare anche codice.
- C'è il concetto di **classi aperte** quindi si possono aprire classi anche già definite in qualsiasi punto e inserire degli altri metodi, anche le classi built-in!
- Solo *false* e *nil* sono valutati come falsi nelle condizioni, qualsiasi altra cosa è valutata come vera.

2.3 Esempi Base

Stringhe palindrome Ruby permette l'utilizzo di caratteri anche non alfanumerici per i metodi, generalmente metodi che ritornano un valore booleano si fanno terminare con ?, mentre metodi che hanno come side-effect quello di modificare l'oggetto sul quale vengono invocati si fanno terminare con !

```
def palindrome?(string)
  string = string.downcase.scan(/[^\W\s_]/).join
  return string == string.reverse
end
```

Si noti il supporto alle espressioni regolari, nativo in Ruby.

Raggruppare anagrammi È un gioco da ragazzi in Ruby combinare le stringhe presenti in un Array in base a quelle che sono anagrammi con altri.

```
def combine_anagrams(words)
  words.group_by{|w| w.downcase.chars.sort.join}.values
end
```

Si vede qui l'utilizzo dei blocchi, questi sono come le lambda functions viste precedentemente in Python

2.4 Metaprogrammazione

La metaprogrammazione è anche conosciuta come metodi che scrivono metodi, vediamo qui due esempi in Ruby.

2.4.1 Esempio di ereditarietà e attr_accessor

Creiamo due classi Dessert e JellyBean che estende Dessert e mettiamoci un po' di cose Ruby-style:

```
class Dessert
  attr_accessor :name, :calories

  def initialize(name, calories)
    @name = name
    @calories = calories
  end

  def healthy?
    return true if @calories < 200
    return false
  end

  def delicious?
    return true
  end
end

class JellyBean < Dessert
  attr_accessor :flavor
  def initialize(name, calories, flavor)
    super name, calories
    @flavor = flavor
  end

  def delicious?
    return false if @flavor == "black licorice"
    return true
  end
end
```

- Il metodo attr_accessor è un metodo che oltre a creare dei campi nella classe, genera anche:
  - Un getter lo stesso nome dell'attributo
  - Un setter (un metodo attributo=) che setta il campo
- Il metodo initialize è il costruttore.
- Si usa @ per accedere ai campi dell'istanza
- le stringhe iniziate da : sono chiamate simboli e si usano quando si vuole sottolineare che quella stringa è un nome di variabile.

2.4.2 attr_accessor_history

Vogliamo ora scrivere un metodo disponibile in tutte le classi che si occupi di generare un setter che mantenga anche la storia dei valori. Vogliamo quindi scrivere un metametodo, cioè un metodo che genererà codice.

Come prima cosa dobbiamo aprire la classe Class per rendere questo metodo in ogni sottoclasse (tutte le classi ereditano da Class).

Ci serviremo anche di class_eval che serve per valutare all'interno della classe del codice che è scritto in una stringa!

```
class Class
  def attr_accessor_with_history(attr_name)
    attr_name = attr_name.to_s
    attr_reader attr_name
    attr_reader attr_name+"_history"
    class_eval %Q{
      def #{attr_name}=(value)
        @#{attr_name} = value
        @#{attr_name+"_history"} = [nil] if @#{attr_name+"_history"}.nil?
        @#{attr_name+"_history"} << value
      end
    }
  end
end

class Foo
  attr_accessor_with_history :bar
end

f = Foo.new
f.bar = 1
f.bar = 2
puts f.bar_history
```

- *%Q{}* è l'equivalente di una stringa doppio-quotata, come *%q{}* è l'equivalente di una stringa mono-quotata, dove si vuole essere liberi di utilizzare dei doppi apici senza dover fare escape degli stessi.
  - Stringa mono-quotata = stringa non mutabile, non valuta le cose al suo interno.
  - Stringa doppio-quotata = permette l'utilizzo di costrutti come *#{var}* per inserire nella stringa il valore della variabile.
- attr_name è una variabile
- attr_reader è un metodo che a sua volta è di metaprogrammazione

2.4.3 method_missing

Quando un metodo non è definito su una classe, quando questo viene invocato, prima di essere passato alla superclasse, viene invocato il metodo *method_missing(method_id)* passando come simbolo il nome del metodo mancante.

In questo modo si possono creare valangate di metodi in maniere semplicissime. Questa è probabilmente una delle cose più belle di Ruby che io sappia fare.

Ad esempio vogliamo aprire la classe Numeric e vorremmo aggiungere tutti dei metodi per fare delle conversioni di valute. Invece che fare ciò possiamo semplicemente utilizzare il metodo method_missing insieme a un dizionario e il gioco è fatto!

```
class Numeric
  @@currencies = {'yen' => 0.013, 'euro' => 1.292, 'rupee' => 0.019, 'dollar' => 1}

  def method_missing(method_id)
    singular_currency = method_id.to_s.gsub( /s$/, '')
    if @@currencies.has_key?(singular_currency)
      self * @@currencies[singular_currency]
    else
      super
    end
  end

  def in(currency)
    singular_currency = currency.to_s.gsub( /s$/, '')
    self / @@currencies[singular_currency]
  end
end

puts 5.dollars.in(:euros)
puts 10.euros.in(:rupees)
```

- @@ indica delle variabili della classe.
- Si usano delle espressioni regolari per rimuovere la s del plurale dai simboli.
- È importante ricordarsi di effettuare la chiamata a *super* in caso se non si decide di intraprendere delle azioni nella method_missing in modo che l'interprete possa continuare a cercare il metodo delle superclassi (infatti se method_missing non viene overridato, la chiamata a super è l'unica cosa che effettua).

2.4.4 Palindromi nelle stringhe e negli enumerabili

Possiamo aggiungere il metodo palindrome? alla classe String in modo che sia accessibile su tutti gli oggetti di tipo stringa.

```
class String
  def palindrome?
    string = self.downcase.scan(/[^\W\s_]/).join
    return string == string.reverse
  end
end

puts "foo".palindrome?
puts "anna".palindrome?
```

In realtà però possiamo estendere il concetto di palindromo a tutti quegli oggetti che sono iterabili. In Ruby il modulo Enumerable è mixato con tutte le classi per le quali si vuole che siano iterabili, quindi possiamo aprire direttamente questo modulo ed inserire qui dentro il metodo:

```
module Enumerable
  def palindrome?
    a = self.map{|x| x}
    return a == a.reverse
  end
end

puts  [1,2,3,4,5]. palindrome?
puts  [1,2,3,2,1]. palindrome?
puts  (1..9). palindrome?
```

3 Conclusioni

Se paragonato ai linguaggi compilati staticamente tipati, come ad esempio il C, la velocità di esecuzione non sono punti di forza dei linguaggi interpretati dinamicamente tipati.

È interessante però notare che esiste il progetto PyPy che è un compilatore di Python scritto in Python e che ottiene delle ottime performances, pur pagando un prezzo in termini di memoria utilizzata; un discorso analogo vale per Rubinius. Comunque, anche in questo modo un qualunque compito che preveda numerosi calcoli puri non è adatto ad un programma Python.

Le performance di Python sono invece allineate o addirittura superiori ad altri linguaggi interpretati, quali PHP e Ruby, e in certe condizioni può rivaleggiare anche con Java.

Non va inoltre dimenticato che sia Python che Ruby permettono di scrivere un'estensione in C o C++ e poi utilizzarla all'interno del codice, sfruttando così l'elevata velocità di un linguaggio compilato solo nelle parti in cui effettivamente serve e sfruttando invece la potenza e versatilità di un linguaggio come Python o Ruby per tutto il resto del software.

Nonostante il discorso delle performances, credo che questo aspetto sia critico solo per una ristretta parte di progetti, ma credo che per la programmazione di tutti i giorni o per lo sviluppo di piccoli tools Python e Ruby garantiscano numerosissimi vantaggi rispetto a C, C++ e Java, infatti permettono di tralasciare tutte delle parti relative a tipi di passaggi, tipi, overload, binding e cose complesse, concentrandosi invece sulle funzionalità di ciò che scriviamo e sapendo che poi ci penseranno i nostri amati linguaggi, seppur un po' più lenti, a far andare tutto per il meglio.